6-25-1987

IPL: Interfaced Prolog/Lisp

Steven J. Recard

Rochester Institute of Technology
School of Computer Science and Technology

IPL: Interfaced Prolog/Lisp

by

Steven J. Recard

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

June 20, 1987

Approved by:

John A Biles 6/25/87

Peter G. Anderson

Name Illegible

Abstract

This thesis report describes the design and implementation of an interface between the two most common artificial intelligence languages, Lisp and Prolog. The interface is accomplished by small extensions to each language, and provides Prolog programs with the capability of invoking Lisp functions. The interface is simple yet powerful; it the supports passing of arbitrarily complex data objects, regardless of data type. The particular language implementations extended were C-Prolog [Pereira,85] and XLISP [Betz,86], both interpreters running under the Unix operating system.

Key words

Prolog, Lisp, Logic programming, Language interfaces, Programming environments, Programming paradigms.

Subject categories

ACM Computing Review:

I.2.5: Computing methodologies .
Artificial intelligence .
Programming Languages and Software

I.2.3: Computing Methodologies .
Artificial intelligence .
Deduction and Theorem Proving

Inspec (IEEE/IEE) Computer and Control Abstracts:

61.40D: High-level languages

61.50C: Compilers, interpreters, and other processors

Acknowledgments

My thanks go to Al Biles whose wise counsel kept this thesis manageable. Without his advice I might still be working on it.

Pete Lutz, the local Unix "guru", deserves special thanks for answering my endless questions. He wasn't officially on my thesis committee, but deserves an honorary position.

My wife cannot be praised enough for her patient encouragement and support. She is an example of the "wife of noble character, worth far more than rubies".

The greatest thanks and praise goes to the Lord Jesus Christ, for none of this would have any meaning without Him. "The Lord is my strength and song; he has become my salvation".

IPL: Interfaced Prolog/Lisp

I hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Steven J. Recard

Steven J. Recard

June 26, 1987

TABLE OF CONTENTS

CHAPTER 1

Summary

This chapter provides a short summary of this thesis report. All information in this chapter was extracted from later chapters. If the reader intends to read the entire report, this chapter can be skipped without loss of information.

There are several programming languages that are used for programming artificial intelligence applications, Lisp and Prolog being the two most common. These two languages are quite different: Lisp provides a functional programming paradigm, Prolog a logic programming paradigm. This report describes a system that combines these two languages by providing a calling interface between them. The resulting interfaced Prolog/Lisp system is called IPL.

IPL was implemented by making source code modifications to XLISP [Betz,86] and C-Prolog [Pereira,85]. Both of these are interpreters running under the Unix* operating system. A "fast prototype" approach was used for the implementation: the two interpreters are run as separate Unix processes with interprocess communications performed by Unix system calls.

IPL provides Prolog programs with the capability to call Lisp functions. (Time pressures precluded support for calls in the reverse direction.) The basic call is provided by the :=/2 predicate. (The notation ":=/2" indicates a structure having two components, with functor ":=".) := is defined as an infix binary operator. Proof of := proceeds as follows. First, the right operand is converted into valid Lisp data structures according to a built-in set of data conversion rules. This converted data is then passed to Lisp where it

*Unix is a trademark of AT&T.

is evaluated. The result of Lisp evaluation is transformed back into Prolog data structures via another set of built-in conversion rules and this data is returned to Prolog. Prolog then attempts to unify this result with the left operand. If the unification is successful, then the proof of := is successful, else the proof fails. := cannot be re-satisfied on backtracking.

The data conversion rules are the heart of the interface. The interface supports most XLISP and C-Prolog data types. Atoms (Prolog constants), integers, and floating point numbers require no conversion. Prolog structures and lists, and Lisp lists and strings, however, require some transformations. The definition of these data conversions is as follows:

(1) A Lisp string maps to a Prolog structure with double-quote as its functor. For example, the Lisp string **"abc"** is equivalent to the Prolog **'"'(abc)**.

(2) A Prolog list maps to a Lisp list with an internal "Prolog type-tag" set to **list**. For example, the Prolog **[a,b,1,2]** maps to the Lisp **(a b 1 2)**, internally tagged as **list**.

(3) A Prolog structure maps to a Lisp list with an internal "Prolog type-tag" set to **structure**. For example, the Prolog **a(b,c)** maps to the Lisp **(a b c)**, internally tagged as **structure**.

In addition to :=, several other extensions were added to Prolog. The predicates :/1, ::/1, and **lisptest**/1 all make calls to Lisp. : and :: are defined as unary prefix operators. The processing for the single operand of each of these predicates is identical to that of the right := operand. The difference arises in the handling of the Lisp evaluation result. :/1 causes the result to be displayed on the user's console. This is useful for interactive Lisp "queries". ::/1 causes the result to be discarded. This is useful for Lisp operations in which only the side-effect of the operation is desired. **lisptest**/1 tests the returned data and succeeds if the returned data is non-nil (not equal to []), else it fails.

To provide notational convenience, several prefix operators have been defined to serve as Prolog equivalents to Lisp read-macros. The Lisp read-macros ' (single-quote),

#' (pound single-quote), ‘ (backquote), , (comma), and ,@ (comma at-sign) may be entered in Prolog as ^ (caret) #^ (pound caret), ‘ (backquote), ',' (comma, appropriately delimited), and ',@' (comma at-sign, appropriately delimited), respectively.

The following are a few examples of IPL usage:

Example 1

Compute the summation of the list of integers in L; unify with the variable **Sum**.

```
?- L = [1,2,3,4], Sum := [apply, #^ +, ^L].
L = [1,2,3,4]
Sum = 10
```

Example 2

Define a predicate that checks to see if a list ends with [ ], i.e. doesn't end in a "dotted pair".

```
islist(L) :- lisptest( [listp, [last,^L]] ).
```

(The Lisp predicate **listp** returns **t** if its argument is a list, else **nil**.)

Example 3

Lisp printing from Prolog.

```
lisp_print(X) :- ::[print, ^X].

?- lisp_print( a(b,[d,e,f(g),h],i) ).
(a b (d e (f g) h) i)
```

The :: causes the **nil** that is returned by **print** to be discarded.

CHAPTER 2

Introduction and background

2.1. Problem statement

There are several programming languages that are used for programming artificial intelligence (AI) applications, Lisp and Prolog being the two most common. Both of these languages are common enough such that reasonably efficient compilers are available for many machines, and there is a growing body of programmers having some knowledge of these languages. Both languages are powerful: any given AI application could use either Lisp or Prolog as the implementation language.

Lisp and Prolog are quite different, however. Lisp is primarily a functional language: all the processing is performed by calling functions. I say "primarily" because all serious Lisp dialects provide the programmer with such "impure" programming constructs as use of side-effects and iteration. Prolog is a declarative language: programming consists of declaring facts and rules, then asking questions. In theory, a Prolog programmer need not be concerned with traditional program control. It is the goal of Prolog to "program in logic", hence the name "Prolog". In practice this goal is not met because of the desirability of such control constructs as the Prolog **cut** operator.

This considerable difference between these two languages is fortunate. Some AI problems are best represented by Prolog's logic programming approach, and others by Lisp's functional approach. Many problems, however, are best solved using both approaches.[1] This need leads to the problem: within one application, how does one make

[1] What may be more desirable is a programming environment supporting multiple programming paradigms, such as is advocated by Brobow [Brobow,85]. For the purpose of this thesis, however, I have limited myself to two paradigms, functional and logic programming, as provided by Lisp and Prolog.

good use of two radically different programming languages?

The simplest solution is to require the programmer to utilize high-level operating system (OS) facilities to implement a Prolog/Lisp interface. This might be done, for example, by structuring the application as a job consisting of several discrete phases, each phase executing either Prolog or Lisp, and operating on a file set. In this example, file I/O would thus provide the basis for the interface.

While this approach will always work, it has several serious drawbacks. First, each programmer is forced to implement his own custom interface. This is wasteful of man-power and may yield a difficult-to-maintain and unreliable interface. Second, this approach is probably machine-inefficient because (a) it employs general purpose, high-level OS facilities, and (b) the interface is outside of the language and thus beyond the scope of the language compiler optimizer. Finally, the resulting interface may be very awkward to use, especially if the particular application being programmed doesn't separate cleanly into a few major "Prolog-processing" and "Lisp-processing" phases.

There are, however, some advantages to using high-level OS facilities:

(1) Lisp and Prolog can be used "as-is," without modification. There are no new languages or special language extensions to learn.

(2) An application programmed in this manner is very portable since the interface is implemented in unmodified Lisp and Prolog. If the language translators can be ported to another system, the application should also port.

I imagine that many applications already use this high-level OS facility approach for interfacing Lisp and Prolog. Probably for that reason, however, the approach is considered too conventional to warrant publishing. I found nothing published on any effort employing such an interface.

Many researchers seem to have decided that it is not possible to provide a good Prolog/Lisp programming environment. Instead, they feel that the proper solution is to design a new language having features of both Lisp and Prolog (and sometimes other

languages).[2] The resulting language, if properly done, has all the functionality of both Lisp and Prolog, but is fully integrated such that it is powerful and easy to use and can be efficiently implemented. Loglisp [Robinson,82] is a good example of such a language.

While such new languages have considerable advantages, they also have one great disadvantage: their newness slows their lack of acceptance. Consider Lisp and Prolog, two fairly well accepted AI languages. Lisp has existed for more than 25 years. Lisp interpreters and optimizing compilers are fairly common, although experienced programmers are not. Prolog is approximately 15 years old. Prolog interpreters and compilers have only recently become available for many computers; optimizing compilers are still rare; and experienced Prolog programmers are rarer still. The point is that there is great reluctance (or call it inertia) within the computer community to accept any new language, regardless of its merits. (This is not to say, of course, that new languages should not be developed.)

Another approach to providing a Prolog/Lisp programming environment is to extend each language to include some reasonable interface to the other. The extension to each language is done in such a fashion that the syntax and semantics of the original language are not changed, except possibly at the interface itself. Existing Lisp or Prolog programs would run unchanged, and programmers need not learn a new language, only the extensions. The interface itself can be crafted into each language's interpreter or compiler, thus the resulting system could be machine-efficient, although possibly not as efficient as a completely new language, such as described earlier.

It is this last approach that I have explored. The remainder of this paper focuses on this interfaced Prolog/Lisp approach.

(Note: The Prolog terminology and syntax used throughout the remainder of this paper is documented in Appendix A. Lisp terminology and syntax is fairly standardized

[2][Barbuti,84; Sato,83; Subrahmanyam,84; Robinson,82; Komorowski,82; Malachi,85; Takeuchi,83; Kahn,81]

and so is not documented in this report.)

2.2. Previous work

Surprisingly, while I am sure that many others have implemented Prolog/Lisp interfaces, almost nothing has been published on this particular approach. D. Bailey's "University of Salford Lisp/Prolog System" [Bailey,85] is the only published work that I could find describing an interfaced Prolog/Lisp system. Bailey describes the reason for the development of this system in this way:

> Rather than invent a new language, with a new syntax incorporating the features of both Lisp and Prolog, it was considered desirable that pure Lisp or Prolog programs should run with little if any modification. [Bailey,85,p.595]

The Salford system is an enviable programming system that supports programming in Lisp, Prolog, and Fortran 77. Lisp and Prolog code may be either interpreted or compiled as required, while Fortran code must be compiled. The system was designed to solve large problems, so run-time efficiency was a goal. It was implemented in an extended version of Fortran 77 and runs on the University's Prime Computers. I shall briefly examine the design of the Salford Prolog-calls-Lisp interface.

In order to implement a Prolog/Lisp interface, it is first necessary to define a mapping between equivalent data objects. Examples of the Salford mapping are [Bailey,85,p.596]:

Prolog	Lisp
fred	fred
noun(table)	(noun table)
age(mary,12.5)	(age mary 12.5)
ZZZ()	(ZZZ)
[alpha,beta,gamma]	("|" alpha "|" beta "|" gamma)

The mapping for the Prolog list **[alpha,beta,gamma]** illustrates a problem that one immediately encounters when attempting to define a Prolog/Lisp interface. Salford has chosen to map simple Lisp lists onto Prolog structures. The consequence is that Prolog

lists now do not have a simple mapping. Bailey points out that if the simple mapping given above is utilized for Prolog lists, then the Prolog list **[alpha,beta,gamma]**, which is actually the Prolog structure **.(alpha, .(beta, .(gamma,[ ])))**, maps to the rather ugly Lisp list **(. alpha (. beta (. gamma ())))**. Bailey admits that "...the result would be a very inefficient representation and would not map into a sensible Lisp structure" [Bailey,85,p.596], hence the reason for the special mapping for Prolog lists.

The Salford system defines no mapping for Prolog variables or Lisp *dotted pairs*, nor was any mention made of Lisp strings.

In the Salford system, Lisp calls from Prolog are accomplished by use of the operators **:=**, **%**, **‘** (grave accent), and **is**. **is** was redefined to allow Lisp to do all arithmetic expression evaluation. **:=** is like **is**, but causes a Lisp global variable to be bound with the result of expression evaluation. For example: [Bailey,85,p.599]

```
pi := 4*ATAN(1).  /* Define the Lisp atomic constant 'pi' */
```

This causes the Lisp atom **pi** to be bound to the value resulting from evaluation of 4*ATAN(1).

The **%** operator allows Lisp functions to be executed for their side-effects without saving the result. Bailey gives the following example: [Bailey,85,p.597]

> The Prolog user can make use of Lisp graphics facilities thus:
> **line(X1,Y1,X2,Y2) :- %MOVETO(X1,Y1),%DRAWTO(X2,Y2).**

Finally, the **‘** (grave accent) operator is used to prevent Lisp evaluation of non-numerical data. Prolog data prefixed with **‘** and passed to Lisp becomes Lisp *quoted*.

Salford Lisp/Prolog has other features I have not described, including a Lisp-calls-Prolog capability, Prolog and Lisp compilation, and a variety of other predefined routines to increase the utility of the system.

As stated before, the Salford system is the only published example of an interfaced

Prolog/Lisp system. Apparently there are some commercial implementations,[3] but these companies have not published their efforts, nor did I have access to their systems. I generally did not attempt to research the commercial market because of the complexities (cost and confidentiality) that would entail.

All the other systems that have been published are not interfaced Prolog/Lisp. Loglisp [Robinson,82], Virginia Tech Prolog/Lisp [Roach,85], TAO [Takeuchi,83], TABLOG [Malachi,85], QLOG [Komorowski,82], and Uniform [Kahn,81], for example, are all new languages. In these languages, at a minimum, the syntax of either Lisp or Prolog has been changed so that the two languages better combine. Some of them, such as QLOG, implement one language via the other. (In practice, Prolog is always implemented in Lisp.) I will not describe these languages because they begin with the very different premise that it is necessary to substantially modify the basic syntax of Lisp and Prolog. I do not intend, of course, to denigrate any of these languages. Some of them appear very sophisticated and powerful. They are, however, in a different class from interfaced Prolog/Lisp systems.

2.3. The scope of my effort

In the early stages of this project I planned to provide a fairly complete integration of Prolog and Lisp. Prolog programs would be able to call Lisp functions, and Lisp programs would be able to make Prolog queries. I planned to support recursive calls between the two languages. I considered run-time efficiency and debugging capabilities. I tried to avoid introducing any unnecessary restrictions. My intent was to produce a useful, user-friendly interfaced Prolog/Lisp system that could compete with AI hybrid languages such as Loglisp. Much of my early design effort was done with these goals in mind.

---

[3] I believe that HP Prolog, from Hewlett-Packard, and LM-Prolog, from LMI, Inc., Los Angeles, CA, provide interfaced Prolog/Lisp, but I did not verify this.

Discussions with my thesis advisor, Mr. John A. Biles, made it clear that the work necessary to achieve these goals was far too great. We therefore decided to limit the effort. Only half of the interface would be done: Prolog could call Lisp, but not the reverse. This would, of course, automatically remove any potential problem of recursive cross-language calls. Run-time efficiency and debugging considerations were made secondary in importance. The emphasis would be placed on defining a Prolog-calls-Lisp interface, and implementing a prototype of that interface.

The particular Prolog and Lisp language translators selected to be interfaced were chosen for reasons of expediency. Both are interpreters: XLISP [Betz,86] and C-Prolog [Pereira,85]. XLISP was chosen because of its small size. It is much smaller than other commonly used Lisp implementations and thus could be more quickly modified while requiring fewer computer resources. C-Prolog was chosen because it was the only commonly-used Prolog implementation available at RIT.[4] Another reason for selecting these particular translators was that RIT had the source code for both, and that each was coded in a common language (the "C" programming language).

I should mention that XLISP, in addition to being a Lisp interpreter, has extensions to support object-oriented programming. I intentionally ignored this aspect of XLISP, as will all references to XLISP throughout the remainder of this thesis.

In the remainder of this report I will use the words “Prolog” and “Lisp” to refer to C-Prolog (version 1.4) and XLISP (version 1.6).

---

[4]Rochester Institute of Technology, Rochester, NY.

CHAPTER 3

Design

3.1. Design Goals

This chapter lists design goals for the interface, presents examples, gives the complete syntax and semantics for the interface, and reflects upon the design. Implementation details are presented in the next chapter.

Listed below are a set of goals that I established while considering various possible designs for the Prolog/Lisp interface. Every potential design was weighed against conformity to these goals. The design selected for implementation best met these goals.

(1) "Keep it simple".

This goal was established for both myself as an implementor, and for the eventual user. It was an important goal for myself to ensure that I would be able to implement in a timely fashion what I had designed. I felt it was a good goal for the user since systems designed with a "keep it simple" philosophy are frequently reliable and easily used.

(2) "The interface should be easy to use and intuitive to understand."

This can be expressed in the negative as "don't get in the way of the programmer". This goal is closely related to the first goal.

(3) "Data transfer between Prolog and Lisp should be *transparent* in the sense that information will not be lost or modified by the interface, regardless of the direction or frequency of transfer across the interface."

I established this goal early in my thesis effort when I thought I would be designing and implementing both the Prolog-calls-Lisp and Lisp-calls-Prolog interface. This

goal would be especially important in such a complete, symmetric system. Even after the decision to do only half the interface, I chose to keep this goal with the hope that in the future someone would eventually add the Lisp-calls-Prolog interface. I also felt that the goal of transparent data flow was conceptually sound.

(4) "It should be possible to pass any data across the interface, regardless of whether the data was compile-time constant or was dynamically created during program execution."

(5) "The presence of the interface should have a minimal effect on existing Prolog and Lisp programs."

This can be rephrased as "don't break working code". As explained in the introduction, it was always my intent to interface Lisp and Prolog, and not to define a new language. This approach would minimize the learning curve for programmers attempting to use the system, and would allow use of existing Prolog and Lisp software.

(6) "The design should allow a machine-efficient implementation."

The wording of this goal is significant. My goal was not to implement a machine-efficient system, only to design a system which would allow a future machine-efficient implementation.

After a complete presentation of the design I will come back to these goals to consider how well they were achieved.

3.2. Introduction to the design

My implementation of interfaced Prolog/Lisp, henceforth called IPL, consists of the capability of a Prolog program to make calls to Lisp functions, passing data to the Lisp functions and having some data returned. The called Lisp functions may also perform useful side-effects such as binding global variables or performing I/O.

It may aid in the understanding of IPL if some of the underlying implementation is understood. IPL is implemented as separate Prolog and Lisp interpreters, each with its own execution environment. As a consequence each has its own symbol name space, thus there is no conflict between Prolog and Lisp program symbol names.

Prolog calls to Lisp are like *top-level* Lisp calls except that the Lisp *read-eval-print* loop becomes just a single *eval*. Otherwise, except for a few additions necessary for the interface, Lisp acts as expected. For example, global variables bound in one Lisp call can be referenced in later Lisp calls because the Lisp environment is preserved from one Lisp call to the next.

Before presenting a detailed definition of the interface, let us first look at a few examples of IPL in use. It is hoped that these examples will convey a general idea of the design. Note that the examples exist only for pedagogic purposes and typically do not illustrate useful operations.

3.3. Introductory examples

In the examples below, computer generated output is shown in ***bold italics*** print.

Example 1

Compute the summation of the list of integers in L; unify with the variable **Sum**. ^ and #^ are the Prolog equivalents to the Lisp ' and #' read-macros.

```
?- L = [1,2,3,4], Sum := [apply, #^ +, ^L].
L = [1,2,3,4]
Sum = 10
```

Example 2

Define a predicate that checks to see if a list ends with [ ], i.e. doesn't end in a "dotted pair".

```
islist(L) :- lisptest( [listp, [last,^L]] ).
```

listp returns t if is a list, else **nil** if not a list.

Example 3

Define a predicate that does a bit-wise AND of the integers in the list L, and unifies the result with R.

```
bitwise_and(L,R) :- R := [apply, #^ 'bit-and', ^L].
```

Example 4

Lisp can evaluate Prolog structures as well as Prolog lists.

```
?- Pi = 3.14159, Ans := sin(Pi*0.5).
Pi = 3.14159
Ans = 1
```

The Prolog **sin(Pi*0.5)** becomes the Lisp **(sin (* 3.14159 0.5))**, which is directly evaluable by Lisp.

Example 5

A Lisp string is represented in Prolog as an atom with functor ".

```
?- A := ['symbol-name', ^car].
A = "car
```

Example 6

Lisp printing from Prolog.

```
lisp_print(X) :- ::[print, ^X].

?- lisp_print( a(b,[d,e,f(g),h],i) ).
(a b (d e (f g) h) i)

?- lisp_print( '"' 'Hello, there' ).
"Hello, there"
```

The :: causes the **nil** that is returned by **print** to be discarded. Must use the prefix operator " to get a Lisp string.

Example 7

Prolog/Lisp interface can pass and return a Prolog structure or list unchanged.

```
?- (a(b,c);d(e)) := ^ (a(b,c);d(e)).
Yes

?- [a,b,c] := ^[a,b,c].
Yes
```

Example 8

The Lisp environment is preserved between calls from Prolog.

```
?- :[progn, [setq,x,^[a,b,c]], [setq,y,x]].
[a,b,c]
Yes

?- :y.
[a,b,c]
Yes
```

More examples are scattered throughout the report. Now let us take a detailed look at the interface design, beginning with what is possibly the heart of the interface, the definition of data conversion between Prolog and Lisp.

3.4. Data conversions

Most XLISP and C-Prolog data types are supported by the IPL interface. Data of the supported data types can be passed in either direction across the interface without loss of information. There are cases in which a supported data type exists in only one of

the two languages. In such cases a data object having such a type will be converted by the interface into some form acceptable to the receiving language. The type information will not be lost, however. The object will still be data typed such that if it returns across the interface, it will be converted back to what it was originally.

In Prolog, this additional type information is encoded using structures with special functors. In Lisp this information is stored internally. Exactly how this type information is stored is discussed later. First let us look at the data conversion rules.

(1) Atoms map to atoms without change. For example:

Prolog	Lisp
abc	abc
'A ;'	|A ;|
[]	nil
t	t

(Note: the vertical bars in |A ;| represent symbol delimiters. Similarly, the single quotes in 'A ;' are symbol delimiters in Prolog used to specify an atom having special characters.)

(2) Integers map to integers without change. For example:

Prolog	Lisp
0	0
1	1
137	137
-5	-5

(3) Floating point numbers map to floating point numbers without change. For example:

Prolog	Lisp
3.14	3.14
.017	.017

(4) Prolog structures map to Lisp lists with

(a) the Prolog functor as the head of the list and the structure's components as the remaining elements of the list, and

(b) the Lisp list tagged internally as "Prolog type = structure".

(This is explained further in section 3.6.1.)

For example (internal type-tagging not shown):

Prolog	Lisp
a(b,c)	(a b c)
+(a,b)	(+ a b)
a(b,c(d,e),f)	(a b (c d e) f)

(5) Prolog lists map directly to Lisp lists, but with the Lisp list tagged internally as "Prolog type = list". (This is explained further in section 3.6.1.)

For example (internal type-tagging not shown):

Prolog	Lisp
[1, 2, a]	(1 2 a)
[car, [quote, [a, b]]]	(car (quote (a b)))
[a | b]	(a . b)
[a, b, c | d]	(a b c . d)

(6) Lisp strings map to Prolog structures with functors " (double-quote). For convenience, " is defined as a prefix operator. For example:

Prolog	Lisp
'"' abc	"abc"
'"' 'A,B'	"A,B"

These data conversions hold for arbitrarily complex data composed of the various types described above. For example, the Prolog structure

[0, a(3.17, [b1, [c1, d(d1, '"' 'A;B'), c2] \| b2], a1), 2]

maps to the Lisp list

(0 (a 3.17 (b1 (c1 (d d1 "A;B") c2) . b2) a1) 2)

with appropriate settings of the internal data type-tags.

Unsupported data types

The following Lisp (XLISP 1.6) data types are not supported by IPL: objects (for object-oriented programming), arrays, file pointers, subrs (built-in functions), and fsubrs (special built-in functions).

The only restriction placed on Prolog data that will be Lisp evaluated is that all variables must be instantiated.

Attempts to pass data of unsupported type will cause a run-time error message to be printed and the Lisp call predicate to fail. For example, the following statements will cause an error:

```
?- A := car.  /* Cannot return the built-in function "car". */
?- A := '*standard-input*'.  /* Cannot return a file pointer. */
?- :[print, ^A].  /* "A" is uninstantiated, thus illegal. */
```

3.5. Modifications to Prolog

The modifications to Prolog consist of the addition of four evaluable predicates to perform Lisp calls, two predicates to provide debugging and Lisp interactive capabilities, one operator for representing Lisp strings, and five unary prefix operators to simulate Lisp read-macros.

3.5.1. :=/2 Lisp call predicate

This built-in evaluable predicate is defined as a binary infix operator. Prolog proof of :=/2 proceeds as follows. First, the right operand is converted into valid Lisp data structures according to a built-in set of data conversion rules. This converted data is then passed to Lisp where it is evaluated. The result of Lisp evaluation is transformed back into Prolog data structures via another set of built-in conversion rules and this data is returned to Prolog. Prolog then attempts to unify this result with the left operand. If the unification is successful, then the proof of := is successful, else the proof fails. := cannot be re-satisfied on backtracking.

The right operand may be any valid Prolog term, except that it may not contain any uninstantiated variables. All variables must be "ground", i.e. instantiated to terms containing no uninstantiated variables. Uninstantiated variables will cause := evaluation to abort with a run-time error message. Recursive (nested) := calls are not supported.

For example:

```
?- [b,c] := [cdr,[quote,[a,b,c]]].
Yes
```

3.5.2. :/1 Lisp call predicate

This evaluable predicate is defined as a unary prefix operator. Like the right := operand, the single : operand is converted and sent to Lisp for evaluation. The result is converted back into Prolog and is displayed on the user's console. : cannot be re-satisfied on backtracking. : is defined by use of :=/2 as follows:

```
':'(LispData) :- Result := LispData, display(Result), nl.
```

3.5.3. ::/1 Lisp call predicate

This evaluable predicate is defined as a unary prefix operator. Like the right := operand, the single :: operand is converted and sent to Lisp for evaluation. The result data is discarded. :: cannot be re-satisfied on backtracking. :: is approximately[1] defined as:

```
'::'(LispData) :- _ := LispData.
```

3.5.4. lisptest/1 predicate

This evaluable predicate provides a convenient way to test the result of Lisp predicates. Like the right := operand, the single **lisptest** operand is converted and sent to Lisp

[1]This definition is only approximate because it would fail if the result of Lisp evaluation were a data type not supported by IPL. For example, given the above definition, the statement ::[setq,stdout,'*standard-output*'] would fail because **setq** would return a file pointer, a data type not supported by IPL.

for evaluation. **lisptest** succeeds if the result of Lisp evaluation is non-nil, else it fails. It cannot be re-satisfied on backtracking. It is defined as:

```
lisptest(LispData) :- Result := LispData, !, Result \== [].
```

3.5.5. lisptrace/1 predicate

This predicate is used for debugging IPL programs. It controls the tracing of Lisp call data. The single component must be one of the following six atoms: **none**, **receive**, **eval**, **both**, **receiveraw**, or **all**; or it may be an uninstantiated variable. If the latter, the variable will be unified with the current **lisptrace** setting.

The six atoms have the following meanings:

none

Display nothing, i.e. turn off Lisp call tracing. This is the initial setting.

receiveraw

Display the "raw" data as received from Prolog. This is used for debugging the IPL interface itself and is not intended for use by the typical user.

receive

Display the data sent from Prolog that is to be Lisp evaluated.

eval

Display the result of Lisp evaluation.

both

Both the **receive** and **eval** data are displayed.

all

receiveraw, **receive**, and **eval** data are all displayed. This is not intended for use by the typical user.

The data is printed to the user's console by the Lisp interpreter using the **print** function. Note that internal list/structure tags are not shown by **print**. A hook exists,

however, for the user to supply an alternate output function. Before the data is printed the interface software checks for the existence of a Lisp function named **lisptrace**. If it exists it will be called with two arguments. The first will be either the (quoted) symbol **receiveraw**, **receive**, or **eval**. The second argument will be the trace data. This function could be used to display the trace data in such a way that the list/structure tags are apparent, or to write the trace data to a file for later analysis.

3.5.6. golisp/0 predicate

This predicate is used for interactive debugging and user initialization of the Lisp environment. When executed, it suspends the Prolog interpreter and switches the user to the Lisp interpreter. The user can then interact with Lisp through the normal Lisp **read-eval-print** loop. Executing the function **continue** will return the user to Prolog.

3.5.7. Prolog substitutes for Lisp read-macros

Five unary prefix operators were defined to substitute for commonly used Lisp read-macros:

Lisp syntax	XLISP function equivalent	Prolog syntax
'expr	(quote expr)	^expr[2]
#'expr	(function expr)	#^expr
`expr	(backquote expr)	`expr
,expr	(comma expr)	',expr
,@expr	(comma-at expr)	',@expr

While these are called unary prefix "operators", they are not Prolog evaluable — they only have meaning to the IPL interface software. Also, while they are defined as unary prefix operators, they can also be entered using structure notation. For example, **^(expr)** is also equivalent to **^expr**. The former may be preferable in certain cases to avoid problems of operator precedence. (A complete listing of all operators, their types

[2]The substitution of ^ for the Lisp single quote is an unfortunate necessity since single quote is already defined in Prolog as an atom delimiter. In theory, single quote could have been used since it is possible to enter an atom consisting of only a single quote. The required notation, however, would be cumbersome: ''''expr

and precedences, is given in Appendix B.)

Since they are Prolog versions of "read-macros", the following will not be true:

```
?- ^a := ^ ^a.
```

Lisp will return **[quote, a]**, not **^a**.

3.5.8. Representation of Lisp strings in Prolog

Lisp has a true string data type and Prolog does not.[3] For this reason, the unary prefix operator " is employed to represent Lisp strings in Prolog. Thus a structure with functor " and a component of a single atom is interpreted by the Lisp call predicates as a Lisp string comprised of the atom itself. For example, the Prolog **'"' 'Hello, world'** or **'"'('Hello, world')** will be mapped to the Lisp **"Hello, world"**. This mapping is symmetric; thus if Lisp returns a string to Prolog, the interface maps the string into a "/1 structure.

The Lisp call predicates require that the component of a " structure be an atom or a variable instantiated to an atom. Any other term will cause the Lisp call predicates to abort with an error message.

"/1 allows the Prolog programmer to easily utilize Lisp functions requiring or returning strings. For example,

```
..., Sym := [gensym, '"'abc], ...
```

generates a new symbol starting with **abc** and unifies it with **Sym**. Another example is

```
/* Whole = Left concatenated to Right. */
cat_atom(Left, Right, Whole) :-
        '"'Whole := [strcat, '"'Left, '"'Right].
```

This clause will, if **Left** and **Right** are instantiated to atoms, unify **Whole** with an atom that is the result of concatenating the instantiations of **Left** and **Right**.

---

[3] C-Prolog supports a notation called strings, but these are only a notational convenience for lists of integers that correspond to ASCII characters. For example, **"abcd"** is actually the list **[97,98,99,100]**.

Prolog parser caveat

The user must be cognizant of Prolog syntax rules when using any of the new Prolog operators. White-space or structure notation is sometimes necessary for correct Prolog interpretation. A few examples will illustrate this:

```
: ^ hello. /* Works OK. Note white-space after : operator. */
:'^'hello. /* This is OK. */
: ^(hello). /* This is OK. */
:^ hello. /* Error —:^ taken as single atom. */
:'"' 'Hello, world'. /* OK. Note white-space after '"' */
:'"''Hello, world'. /* Error: Lisp atom will be |"'Hello, world| */
```

3.6. Modifications to Lisp

The changes to Lisp were few. Two built-in functions were added that operate on some new internal data structures, and a few other functions were added for user convenience. Now let us look at the exact definition and purpose of these functions.

In IPL, the Prolog structure **a(b,c)** and the list **[a,b,c]** are represented in Lisp as the lists **(a b c)** and **(a b c)**. This, of course, would be ambiguous if there were no way to determine which was a list and which was a structure. This leads us to the new built-in functions **p$type** and **p$settype**. These functions operate on Lisp internal *type-tags*, which specify whether a Lisp list is tagged as type **list** or **structure**. In the example above, the first Lisp list would be tagged as type **structure**; the second list would be tagged as type **list**. Section 3.6.1 gives more details on how type-tags are implemented.

The type-tags are manipulated by the Prolog/Lisp interface. Data passed from Prolog to Lisp that results in creation of a Lisp list is always tagged. This action is reversed for lists returned to Prolog; the interface interrogates the tags to determine what Prolog structures must be built.

The function **p$type** reads the type-tags. It takes a single argument, which must evaluate to a Lisp list, and returns either the atom **list** or **structure**. The function **p$settype** modifies these tags. It takes two arguments, a Lisp list and a Prolog-type specified by the atoms **structure** or **list**, and sets the list type-tag as specified. Both input arguments are evaluated.

Lisp lists not created by the Prolog/Lisp interface default to the Prolog type **list**.

Here is a summary of the definition of these functions:

(p$type <list>)

<list> must evaluate to a Lisp list (possibly **nil**).

Returns the atom **list** or **structure**.

(p$settype <list>, <type-specifier>)

<list> must evaluate to a Lisp list (possibly **nil**);

<type-specifier> must evaluate to either the atom **list** or **structure**.

Returns <list>, appropriately type-tagged.

(p$settype nil, 'structure) will fail with an error message.

Examples:

(p$type x)

will return **list** if **x** is bound to the list **(a b c)**, which had been **[a,b,c]** in Prolog.

(p$type (cadr y))

will return **structure** if **y** is bound to the list **(a (b c) d)**, which was **[a,b(c),d]** in Prolog.

(p$type (quote (a b c)))

will return the atom **list**.

(p$type nil)

will return the atom **list**.

(p$type 'abc)

will cause an error.

(progn (setq x '(a b c))

(p$settype x 'structure))

will cause the list bound to **x** to be set to Prolog type **structure**.

Returns **(a b c)**.

For convenience, two other functions are defined as follows:

(defun p$setstruct (arg) (p$settype arg 'structure))

(defun p$structp (arg) (eq (p$type arg) 'structure))

(p$setstruct arg) sets **arg** to be of type **structure**. **(p$structp arg)** is a Lisp predicate function that returns t if arg is tagged as **structure**, else it returns **nil**.

3.6.1. Type-tags

Type-tags allow a Lisp list to be marked as being of Prolog type **list** or **structure**. Every Lisp list node ("cons cell"), regardless of its origin, is type-tagged as either **list** or **structure**. Initially, all nodes default to being tagged as **list**. The functions **p$settype** and **p$type** can be used to change or read, respectively, a node's type-tag. The Prolog/Lisp interface software internally uses these same functions.

For example, the data structures resulting from the evaluation of **(setq y '(a (b c) d))** is shown in Figure 1.

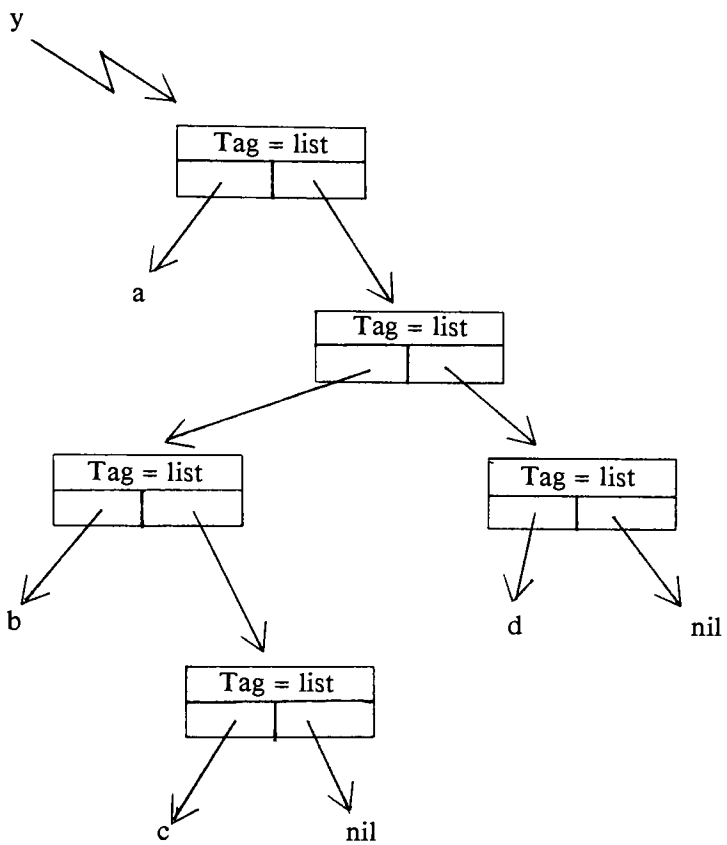


Figure 1

If the statement **(p$setstruct (cadr y))** is now evaluated, then the sub-list **(b c)** will be tagged **structure**. This is shown in Figure 2.

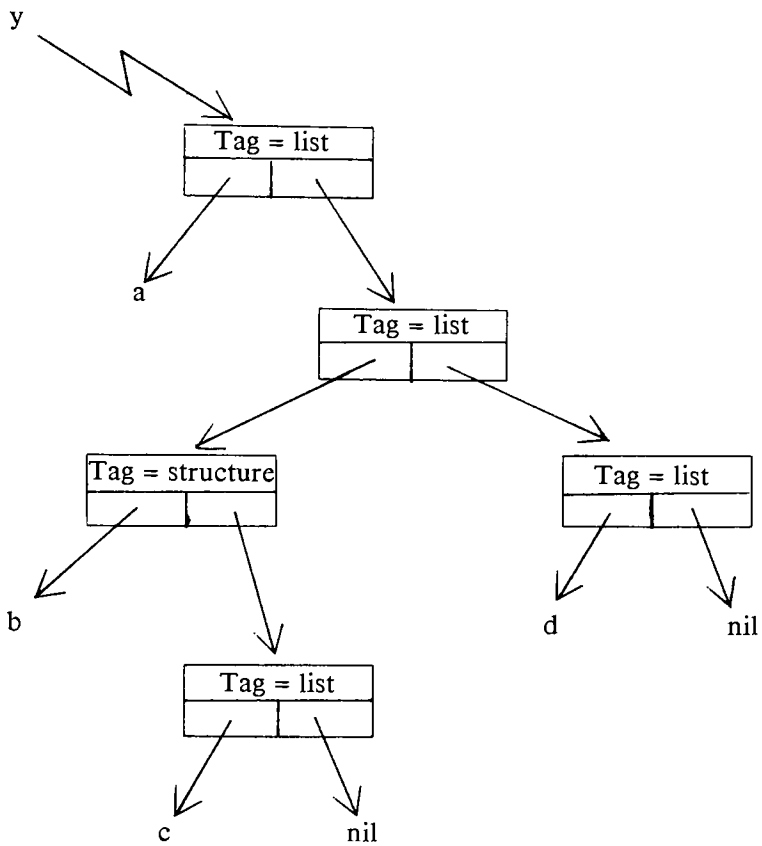


Figure 2

The Prolog equivalent to each of these data structures is **[a, [b, c], d]** for the first, and **[a, b(c), d]** for the second.

It is possible for the user to create type-tagged lists that have no equivalent Prolog representation. For example, while the following statements are evaluable, the resulting data structures cannot be passed to Prolog:

(p$setstruct '(a)) ; Structure must have >0 components.

(p$setstruct '(5 a)) ; Functor cannot be a number.

(p$setstruct '((a b) c d)) ; Functor cannot be a list.

(p$setstruct '(a b . c)) ; Cannot have a dotted pair within a structure.

3.7. Comments on the design

I considered many designs before settling on the particular approach used in IPL. This section first examines several of these alternative designs and shows why they were rejected. Next, IPL is compared to similar extant systems. Finally, I reflect upon IPL, considering to what degree it achieves its design goals.

3.7.1. Alternative designs

3.7.1.1. Initial approach

IPL in its final form is substantially the same as it was when first proposed. It did, however, undergo some evolution. The original approach defined a different Lisp call interface. The predicate **lisp$**/1 was defined as having functional semantics. During Prolog inference, any occurrence of a **lisp$**/1 structure would be replaced by the result of Lisp evaluation of the single component. For example, consider the following code fragments:

,..., A = lisp$([quote, hello]),...

,..., A = [a,b,c], abc(lisp$([car,[quote,A]])),...

,..., call(lisp$(...)),...

This functional approach did not fit well into Prolog's declarative semantics. A new approach, embodied in the :=/2 predicate, was chosen. :=/2 was modeled after other meta-logical predicates such as =../2. This new approach was easier to implement and was more consistent with the rest of Prolog.

3.7.1.2. Alternative approaches to data conversions

Many designs were considered while attempting to solve the difficult problem of passing data between Prolog and Lisp. This is a recognized problem by anyone who has tried combining the two languages. Brobow [Brobow,85,p.1402] says:

> ...one would like to see smooth transitions when calling in both directions. [...] Calling between such Lisp and Prolog implementations, even if they exist in the same environment, requires an awkward transformation of data structures.

The problem occurs for those data types that aren't common between both languages: Prolog has lists and structures, but Lisp has only lists; Lisp has strings, Prolog does not. First let us look at different approaches to solving the list/structure problem.

Alternative solution #1

Possibly the most obvious solution to the list/structure data-passing problem is to rely on the fact that Prolog lists are actually ./2 structures. For example, the list **[a,b,c]** is actually the structure **.(a, .(b, .(c,[ ])))**, where **[ ]** is equivalent to the Lisp **nil**. Knowing this, the mapping becomes simple: all Prolog structures (including lists) map to Lisp lists. For example,

Prolog	Lisp
a(b,c)	(a b c)
[a,b,c]	(. a (. b (. c ())))

(There are obviously other mappings, such as **a(b,c)** to **(a (b c))**, but they are not significantly different.) This approach has the advantage that it is conceptually simple and easy to implement. It has serious disadvantages, however. It results in very awkward and unnatural Lisp lists. These lists will either have to be "normalized" via preprocessing, thus losing the type information and incurring run-time overhead, or all Lisp routines will have to be sensitive to such lists. Neither are very attractive options. The Salford Lisp/Prolog System [Bailey,85] considered but rejected this approach.

It is easy to illustrate the drawbacks of this approach. For example, the Prolog

L = [a,b], ... , :[car, [quote, L]]

will result in the Lisp

(. car (. quote (. a (. b ()))))

which obviously isn't evaluable. It must thus be coded in Prolog as **car(quote(X))**, which becomes the Lisp **(car (quote (.a (.b nil))))**, which evaluates to ".". Thus, from Prolog, the **car** of the list **[a,b]** isn't **a**, it is "." !

Another example is that **length(quote([a,b,c,d]))** doesn't return "4"; it returns "3". (The Lisp function **length** returns the number of elements in a list.)

These drawbacks recur in varying ways in all of the alternative designs that inject the type information directly into the Lisp lists. The result is systems that violate many of the aforementioned design goals.

Alternative solution #2

The designer may be tempted to ignore the list/structure problem and force the user to supply any necessary disambiguating type information. The designer may, for example, allow both of the Prolog terms **a(b,c)** and **[a,b,c]** to map to the Lisp list **(a b c)**. Realize that this means that Lisp can return either lists or structures to Prolog, but not both. The system will be designed to return either one or the other. The user will have to be cognizant of this and may have to perform his own data transformations to force the data to the proper type.

While this approach is simple to implement, it puts a considerable burden on the user. With regards to my design goals, it results in a system that is difficult to use and data transfers that are non-transparent.

Alternative solution #3

Another approach that is typically considered is to utilize "reserved" Lisp atoms to indicate data type. For example, the Prolog list **[1,2]** could be mapped to the Lisp list **(. 1 2)**, or **(list 1 2)**, or **(nil 1 2)**. Similarly, the Prolog structure **a(b,c)** could be

represented in Lisp as **(struct a b c)**, or simply **(a b c)** if Prolog lists were uniquely identifiable.

This approach has the same disadvantages as the first alternative approach: the resulting Lisp data structures are awkward to manipulate because the type information is too visible. Another problem is that certain atoms become "reserved", so use of these atoms must be avoided. This could be very difficult if the data being passed originated external to the program.

Alternative solution #4

When attempting to resolve the list/structure problem the designer may decide to use Lisp property lists to indicate Prolog type. One way to do this is to tag the first element of the Lisp list with a property indicating **list** or **structure**. For example, the Prolog structure **a(b,c)** would map to the Lisp **(a b c)** with atom **a** having property **Ptype = structure**; the Prolog list **[d,e,f]** would map to the Lisp **(d e f)** with atom **d** having property **Ptype = list**.

The problem with this alternative solution lies in the semantics of Lisp atoms and property lists. Consider the Prolog term **a([a,b])**. This could be represented in Lisp as **(a (a b))**. Internal to Lisp there is only a single atom **a**, and it has a single property list. Thus atom **a** could have either property **Ptype = list** or **Ptype = structure**, but not both.

Alternative solution #5

I briefly considered an approach employing dual lists, one containing the data and one giving the type information. For example,

Prolog	Lisp
[1,2,3]	((1 2 3) (list int int int))
a(b,[d(e),f])	((a b ((d e) f)) (struct functor atom (list (struct functor atom) atom)))

This has the advantage that the data being passed is not cluttered with type information. It has many disadvantages, however. First, how is the type list processed by Lisp? Either the user must supply some Lisp routine that removes and separately handles the type list, or the Lisp evaluator must be changed. Second, the addition of this type list will cause increased usage of computer resources.

String alternative solutions

Strings were the other area of C-Prolog and XLISP data type incompatibility. Aside from the approach I ultimately selected, I considered only one other approach: representing XLISP strings as Prolog atoms that start and end with double quotes. For example, the Lisp string **"abc"** would be represented in Prolog as the atom **'"abc"'**. (The single quotes are Prolog syntax for symbol delimiters.)

While this solution is workable, it has several drawbacks. First, the double quote character becomes "reserved".[4] One can no longer pass a Prolog atom that is delimited by quotes. For example, the Prolog **atom '"abc"'** cannot be passed to Lisp to become the Lisp **atom |"abc"|**. One must also decide on the semantics of such Prolog atoms as **'"ab'** or **'"ab"cd"'**.

A second problem with this approach to strings is that the interface will be less run-time efficient. The interface will have to parse all atoms to determine if they are

[4]Admittedly, with the IPL approach to strings, the functor " is reserved in a similar sense.

delimited by matching quotes.

Another problem is the difficulty in Prolog of manipulating atoms as character strings. For example, consider the following Prolog code, which utilizes the IPL approach to string handling:

```
/* Suffix the Singular_atom with an 's'. */
pluralize(Singular_atom, Plural_atom) :-
        '"'Plural_atom := [strcat, '"'Singular_atom, '"'s].
```

Consider the Prolog code necessary to perform the equivalent actions if the alternate string approach was employed! The **name/2** predicate would have to be used to explode the atom, the double quotes would have to be prefixed and suffixed, and so forth.

3.7.2. IPL compared to extant systems

List processing

As I stated at the beginning of this report, the University of Salford Lisp/Prolog system is the only published system whose design allows easy comparison to IPL. The Salford system, however, is a serious programming system, which undoubtedly took considerable effort to develop. From that perspective it is much advanced over IPL. Let us, however, compare the common portions of the two systems, the Prolog-calls-Lisp interface design.

The Salford system and IPL use a similar mapping for Prolog constants and structures. Their mapping for Prolog lists is quite different. They map the Prolog list **[a,b,c]** to the Lisp list **(a '|' b '|' c)**. This approach is a variant of the third alternative approach presented earlier in which "reserved" Lisp atoms are used to indicate data type. As explained before, this approach results in very awkward and unnatural lists. Such a list would typically have to be preprocessed (thus wasting processor time) before being applied to any function. For example, in IPL it is trivial to write a Prolog clause that uses Lisp to compute the average of a list of integers:

```
average(List,Ave) :-
        Ave := [/, [apply, #^(+), ^List], [length, ^List]].
```

This code is very straightforward with no complications introduced by IPL. It takes advantage of the built-in Lisp functions **apply**, **+**, and **length** . It would be much more difficult to do this in the Salford system: either the "|" 's would first have to be removed, or the programmer would be forced to construct his own averaging function that was cognizant of the "|" 's. Either way, the code would be more complex and less intuitive, with more machine resources required to process the list.

"is" semantics

The Salford system allows Lisp to do all the arithmetic evaluation for the Prolog **is** operator. Thus in the Prolog phrase **Y is X*2+1**, the **X*2+1** is evaluated by Lisp. This is easily done since Prolog internally stores **X*2+1** as **+(*(X,2),1)**, which Salford then converts into the Lisp list **(+ (* X 2) 1)**, which can be directly evaluated by Lisp. Similarly, this can be done in IPL by **Y := X*2+1.**

Other differences

There are several other minor syntactic differences between IPL and the Salford system. The IPL quote operator ^ is ‘ (backquote) in Salford. The Salford Prolog % prefix operator is equivalent to the IPL :: operator. The Salford infix := operator is a shorthand for the Lisp **setq** function. For example, the Salford statement

```
pi := 4 * ATAN(1).
```

can be coded in IPL as

```
::[setq, pi, 4*ATAN(1)].
```

If desired, an IPL ::= binary infix operator could be defined to behave identically:

```
:- op( 925, xfx, '::=' ).
::=(Sym,Eval) :- ::[setq, Sym, Eval].
```

I should reiterate that I do not feel that IPL is generally superior or even comparable to the University of Salford system. The latter is a commercial quality product that

has a very efficient machine-level implementation, supports Lisp calls to Prolog, supports Fortran 77, and provides Lisp and Prolog compilation. There are aspects of the design, however, in which I feel IPL is equal, and in some cases superior.

3.7.3. Degree of goal satisfaction

Section 3.1 contained design goals that I set before beginning the design of IPL. Let us now briefly reflect on the degree to which IPL satisfies these goals.

The first goal was to "keep it simple." While it is admittedly difficult to quantify simplicity, I feel that IPL is quite simple. A few items were added that were theoretically unnecessary but provided notational convenience.

It was partly because of this goal that I decided not to support Lisp syntax within Prolog programs. The unfortunate consequence is that Lisp calls from Prolog have a "Prologized" syntax. The advantage of utilizing only Prolog syntax is implementation simplicity: I didn't need to modify the Prolog parser. All parsing of Prolog code is done by the (unmodified) Prolog parser. Another advantage of this approach is that the "pass any data" goal is much more readily achieved since this approach relies on existing Prolog data structures.

The second goal was to provide an intuitive and easily used interface. Like the first goal, this is difficult to measure. I am generally pleased with IPL with regard to this goal, except for the necessity of "Prologized-Lisp" syntax.

I was especially pleased with the satisfaction of the goals "transparent data flow" and "pass any data". All commonly used data types in C-Prolog and XLISP are supported by IPL.

Another stated goal was to "minimize the effect on existing software". This goal was accomplished by defining a narrow interface employing only a few predicates/functions, thus introducing only a few new reserved words. Aside from these few additions, Lisp and Prolog were left substantially untouched. This goal was also sup-

ported by the Prolog/Lisp system employing separate name spaces, thus eliminating, for example, inadvertent conflict between variables in Lisp and Prolog programs.

The final goal was a design that would allow a machine-efficient implementation. Compared with the alternative designs that were considered, I am pleased with the potential efficiency of this design. As I mentioned before, while the design allows for a machine-efficient implementation, I did not take advantage of this, but rather employed a "fast prototype" approach that allowed me to quickly test the design.

In summary, I feel the design goals were largely met. They were not perfectly achieved, but this is reasonable since some of the goals are conflicting, and since Lisp and Prolog are diverse languages and thus not readily interfaced.

CHAPTER 4

Implementation

4.1. Implementation goals and methodology

Several goals guided the implementation effort. The first was to employ a methodology that would minimize the implementation effort. This allowed me maximal time for building and refining a working system at the expense of a less machine efficient implementation. The second goal was to minimize the changes to the XLISP and C-Prolog interpreters, which would in turn maximize the portability and maintainability of the resulting system. The third goal was similar to the second. It was obvious that some changes to both XLISP and C-Prolog would be necessary. The third goal was to make these changes in such as way that XLISP and C-Prolog would still be usable stand-alone. The rationale behind this was to minimize system maintenance — single copies of complex software are more easily managed than multiple copies.

A "rapid prototyping" methodology was utilized to satisfy the goal of minimal implementation effort. I made maximal use of O.S. facilities and high-level languages (Prolog and Lisp) and minimal use of "low-level" languages ("C"). The majority of the Lisp/Prolog interface software was coded in Lisp and Prolog. "C" code was only written when it was necessary to manipulate data structures and resources internal to the interpreters. This methodology and the resulting heavy use of Lisp and Prolog as implementation languages also helped satisfy the other goals.

4.2. Approach

4.2.1. Overview

IPL is implemented as two separately scheduled Unix processes communicating via Unix pipes. The modified C-Prolog interpreter, henceforth called *xprolog*, is the parent process. The modified XLISP interpreter, named *pxlisp*, is the child process. Typically the user only interacts with xprolog, communication with pxlisp being handled internally by xprolog. However, as will be seen later, direct user communication with pxlisp is also possible. Figure 3 shows the basic IPL configuration.

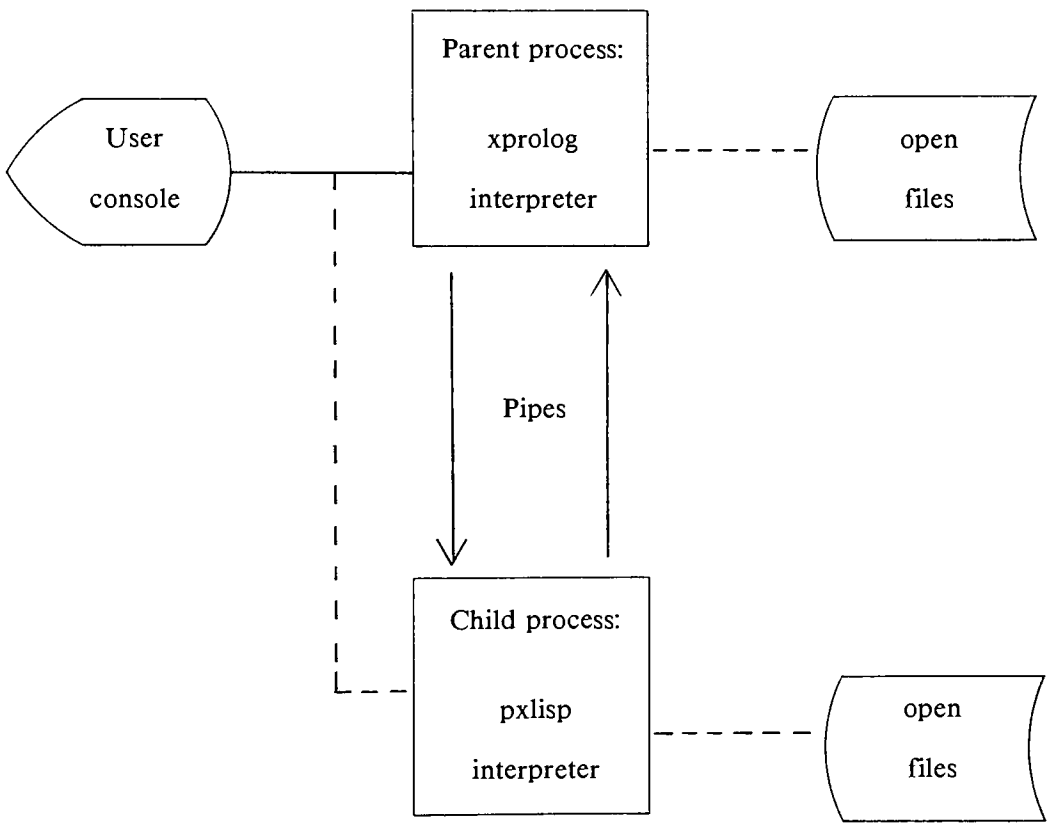


Figure 3
IPL processes and communication channels
(Dotted lines indicate optional connections.)

When the pxlisp process is created, the reserved symbols ***standard-input*** and ***standard-output*** are initialized to the file pointer for the user's console. Lisp I/O directed to either of these "files" will thus result in direct I/O to the user's console. Aside from explicit user programming, the only time IPL will cause pxlisp I/O to the user console is as a result of the **golisp**/0 or **lisptrace**/1 Prolog predicates, or Lisp evaluation errors.

xprolog is designed such that the pxlisp process is not created until it is needed, thus minimizing system overhead. The pxlisp process will automatically be created upon the first evaluation of a Lisp call predicate (:=, :, ::, **lisptest**, **golisp**). It will then remain until the user exits xprolog, although if desired, the user can kill the pxlisp process at any time by evaluation of the Prolog goal **close($lispio)**.

Unix pipes serve as the communication channels between the two processes. The only exception is that xprolog uses a Unix *signal* to kill pxlisp. All Lisp call and result data is sent on these pipes. All data is packetized before transmission. Each packet contains the data to be transmitted plus control information.

Every Lisp call results in a sequence of processing steps. First, the data to be evaluated is converted into a Lisp s-expression. Then it is packetized with control information and sent via pipe to the pxlisp process. xprolog awaits the return packet by reading from its "receive pipe". It will be suspended until the packet arrives.

While xprolog is executing, the pxlisp process is suspended awaiting a packet on its receive pipe. pxlisp will execute as soon as it receives a packet. Once received, the packet is disassembled and all data stored. The data to be Lisp evaluated is first preprocessed to ready it for final Lisp evaluation. This preprocessing step sets any internal list/structure tags, and creates any Lisp strings. The preprocessed data is then evaluated by the usual Lisp **eval** function. The result is converted back into Prolog data structures, packetized with control information, and sent back via pipe to xprolog. pxlisp then returns to reading its receive pipe and is suspended.

xprolog now receives the result packet and resumes executing. The packet is disassembled and the result data used as appropriate for the particular Lisp call predicate being evaluated.

Section 4.2.3 describes packet formats. Section 4.2.4 gives an illustration of the processing steps resulting from a Lisp call from Prolog.

4.2.2. Error handling

Data conversion errors

A Lisp call results in two separate data conversion steps, first from Prolog syntax to Lisp syntax, and then later from Lisp back to Prolog. Errors may be encountered during either of these conversion steps. An error will occur if the user attempts to send some data object that is either (1) not supported by IPL, such as Lisp arrays, or (2) of an illegal type or form for the receiving language, such as Lisp returning to Prolog a structure having a numeric functor.

Section 4.2.1 described the separate steps of data conversion, packetizing, and transmission of the converted data via pipe. In practice, for reasons of run-time efficiency and programming simplicity, these steps are not done separately but rather are done in parallel. The data is sent out on the pipe as it is converted. This approach, however, complicates error handling: How does one handle the situation in which some data has been sent, and then an error is encountered? For example, an uninstantiated variable may be nested deep within a Prolog list that is to be Lisp-evaluated. The beginning of the list will already have been sent when the uninstantiated variable is encountered. The data already sent can't be recalled — it may have already been read by pxlisp.

The solution adopted was to continue conversion and transmission, but to substitute a special "error symbol" for the offending illegal data, and to set an "error found" control flag at the end of the packet. This flag is the last data item in the packet, thus it can always be set if an error was previously encountered. Upon receiving a packet, a process

first checks this flag. If it is set, then the received data is in some way faulty. In this case pxlisp will echo back the faulty data to xprolog without further processing. xprolog, upon receiving such a packet, will indicate an error and will display it on the user's console.

This approach is simple and effective, and allows the error handling processing to be coded entirely in Lisp and Prolog without requiring any modification of the interpreters.

Lisp evaluation errors

Errors may also occur during Lisp evaluation. pxlisp traps all such errors using the Lisp **errset** function. The normal Lisp error message is printed followed by an IPL message ("Lisp evaluation error trap!"), then the offending data is returned to Prolog with the error found control flag set.

4.2.3. Packet formats

There are only two packet formats: one for xprolog to pxlisp and one for the reverse direction. Both packets are actually lists in the receiving language. This simplifies the job of packet assembly and disassembly.

xprolog-to-pxlisp packet format

The xprolog to pxlisp packet format is as follows:

(<Lisp query data> <trace flag> <return data flag> <error flag>)

<Lisp query data> is a specially processed version of the data to be evaluated by Lisp. It contains within it the Lisp code to create Lisp strings and to set any list/structure tags. If the **receiveraw** mode of **lisptrace** is set, pxlisp will display this data exactly as received. Lisp must evaluate <Lisp query data> twice in order to get the result that will be sent back to Prolog. The first evaluation builds strings and creates lists with the list/structure tags set. The second evaluation produces the result, which may include side-effects.

For example, if the Prolog statement **: ^[a, sin(5), c]** is evaluated, then <Lisp query data> will be

‘’(a ,(p$setstruct ‘(sin 5)) c),

which after first evaluation will yield **(quote (a (sin 5) c))**, in which the list **(sin 5)** is tagged as **structure**.

<trace flag> is set according to the current setting of **lisptrace**/1. It will be either **receiveraw**, **receive**, **eval**, **both**, **all**, or **none**. Lisp uses this to determine what, if any, data is to be displayed to the user's console (or output via the optional Lisp **lisptrace** function).

<return data flag> indicates whether or not the result of final Lisp evaluation is to be returned to Prolog. The Prolog **::** sets this flag to **nil**; all other Lisp call predicates set it to **t**.

<error flag> is either t or **nil** to indicate whether or not an error was found during data conversion. The use of this flag is described in Section 4.2.2, "Error handling".

pxlisp-to-xprolog packet format

The pxlisp to xprolog packet format is quite simple, containing only two items:

[<return data>, <error flag>]

<error flag> is analogous to the similarly named component of the xprolog-to-pxlisp packet. **t** means error found, **[]** means no error found. It will be set (to t) if either a Lisp-to-Prolog conversion error was found or if <error flag> in the xprolog-to-pxlisp packet was set.

<return data> will be one of three possibilities. If the <return data flag> in the xprolog-to-pxlisp packet was **nil** then <return data> will always be **[]**. If <error flag> is set then <return data> will be the data received from xprolog. Otherwise, <return data> will be the result of final Lisp evaluation.

4.2.4. Illustration of a Lisp call from Prolog

In the following example, comments are shown in *italics*.

Lisp call from Prolog:

```
?- A := ^[a, b(c), d].
```

xprolog-to-pxlisp packet:

```
(
   ''(a ,(p$setstruct '(b c)) d)    ; query data. ' = backquote, ' = quote.
   none                             ; trace flag = no tracing.
   t                                ; return data flag = yes, return it.
   nil                              ; error flag = no error found.
)
```

Result of pxlisp evaluation:

```
(a (b c) d)                         ; The list (b c) is tagged as structure. (See section 3.6.1)
```

pxlisp-to-xprolog packet:

```
[
   [a, b(c), d],                    /* return data. */
   [ ],                             /* error flag = no error found. */
]
```

Final xprolog response:

```
A = [a, b(c), d]
```

4.2.5. Modifications to C-Prolog interpreter

While most of IPL was programmed in the Lisp and Prolog languages, some modifications to the interpreters were necessary. In the case of C-Prolog, most changes were related to initialization of and I/O with the pxlisp child process. One result of these changes was the definition of the special atom **$lispio**. The semantics of **$lispio** are like those of the atom **user**. For example, **tell(user), write(abc)** results in **abc** being displayed on the user's console. **tell($lispio), write(abc)** results in **abc** being piped to pxlisp. **see($lispio)** results in reading from the pipe, and **close($lispio)** results in the pxlisp process being killed. The first **see** or **tell** to **$lispio** causes the creation to the pxlisp process.

A **$flush**/0 built-in predicate was added, which causes any buffered output data to be written to the current output stream. This was necessary since, for efficiency reasons, output to the pipe is buffered.

The built-in predicate **$writelq**/1 was added to write atoms "quoted" as necessary for pxlisp. It is analogous to the Prolog predicate **writeq**/1. For example, **$writelq('ab|cd\ef')** will output **|ab\|cd\\ef|** to the current output stream. This predicate could have been coded in Prolog, but the performance penalty would have been unacceptable.

4.2.6. Modifications to XLISP interpreter

The key modifications to XLISP were the additions to support list/structure tagged lists. These changes were made without requiring any additional node (i.e., "cons cell") space and with no measurable affect on interpreter execution speed. As described in Chapter 3, the Lisp functions that manipulate these tags are **p$type** and **p$settype**.

The function **$flusho** was added, which causes any buffered output data to be written. **($flusho [<sink>])** will flush the output buffers for <sink>. ***standard-output*** will be assumed if <sink> is omitted. Like the xprolog **$flush**/0 predicate, **$flusho** is used when writing to the pipe.

($unbuf [<sink>]) causes all I/O to <sink> to be unbuffered. ***standard-output*** will be assumed if <sink> is omitted. This function should only be used on I/O sinks that have been opened but have not yet been read or written.

$unbuf is used to cause all output to ***standard-output*** (the user's console) to be unbuffered. If this were not done then pxlisp output to ***standard-output*** would not work properly on some Unix systems.[1] This is admittedly not machine-efficient, but was

[1]For example, unbuffered output was not necessary for the system on which most of the development was done, an AT&T Unix PC, version 3.0. It was necessary, however, when IPL was ported to a Pyramid 90X — OS/X. Additionally, if buffered output to ***standard-output*** is desired, only a single call to **$unbuf** in the file "pxlinit.lsp" need be deleted.

tolerated since the amount of output to the user's console is typically small.

$pprinc is the Lisp analog to the Prolog **$writelq**/1 predicate. **($pprinc '|ab'cd|)** will output **'ab"cd'**, as required by Prolog.

Like Common Lisp, XLISP 1.6 converts all symbols to upper-case upon input. I removed this conversion and made pxlisp case-sensitive. This was the simplest and most run-time efficient solution to what would otherwise have been an IPL problem of case-sensitivity. For example, consider the following Prolog query:

?- **a := [quote,a]**

It is obviously desirable that the **a** in **[quote,a]** be kept in lower-case, but what about **quote**? A lower-case **quote** would probably fail in a Lisp environment expecting all symbols to be in upper-case. A similar problem would exist with Lisp keywords such as **:test**, **&aux**, or even **car** and **cdr** as **setf** place specifiers.

The remaining changes were minor: XLISP was enhanced to support the |symbol| notation for symbol delimiters, a "-q" command line option was added to allow pxlisp to start up quietly, and a "-i" option was added to allow pxlisp to ignore interrupt and quit signals.

4.3. Comments

The implementation goals and resulting approach served very well for developing a prototype system and accelerating all phases of the effort. Most of the coding was done in Prolog and Lisp, with some Unix system calls. Appendix C contains almost all the Lisp and Prolog code — less than twenty pages.

Testing was simplified by the use of the interpretive high-level languages. It was also made easier by careful system design, which allowed the individual processes to be tested stand-alone before being tested together. Most of the IPL interface software could be tested without requiring that both parent and child process be present.

CHAPTER 5

Conclusions

5.1. Possible future extensions and future thesis topics

There are many extensions that could be added to IPL. The most obvious and probably most desirable is support for Prolog calls from Lisp. This capability was originally planned but was then set aside to reduce the effort. Some thought had been given to the interface before it was put aside, and I will mention a few of my ideas for possible future use.

I envisioned a Lisp function that would take one argument, apply the data conversion rules, pass the data to Prolog, and let Prolog attempt to "prove" the passed data. The returned data would be one of three possibilities: (1) **nil** for failure to prove the goal, (2) **t** for a successful proof but no variable instantiations, or (3) a list of dotted pairs for a successful proof with variable instantiations. For example, to attempt to prove the Prolog goal

parents_of(jacob, Mom, Dad)

the following Lisp call would be made:

(prolog$ (p$setstruct '(parents_of jacob Mom Dad)))

Depending upon the contents of the Prolog database, the return data might be, for example, **nil** or **((Mom.rebekah) (Dad.isaac))**

The Lisp backquote and comma read-macros could be useful if Lisp evaluation was necessary to form the goal. For example, if the atom **child** was bound to the value **jacob**, then the call to Prolog could be coded thus:

```
(prolog$ (p$setstruct '(parents_of ,child Mom Dad)))
```

Notational convenience could be provided by adding a read-macro to replace the **p$setstruct** call, and by providing functions that would accept multiple goals and build them into a Prolog conjunction structure.

The addition of the Lisp-calls-Prolog interface will force the designer to resolve the issue of recursion. If allowed, the designer will have to decide upon the semantics of Lisp-Prolog-Lisp or Prolog-Lisp-Prolog calls.

Aside from the Lisp-calls-Prolog interface, other areas for exploration are:

(1) Change the current design to support passing of Prolog uninstantiated variables.

(2) Support more programming paradigms. For example, XLISP supports object-oriented programming. This could be included into IPL, as could interfaces to C, Unix, etc.

(3) Support Lisp syntax within Prolog to eliminate the "Prologized Lisp" syntax.

(4) Replace XLISP with a "real" Lisp implementation such as Franz-Lisp or Common-Lisp. Extend the interface to support some or all of the data types present in the new Lisp dialect.

(5) Replace the current prototyped IPL implementation with a more machine-efficient implementation.

(6) Investigate data sharing between XLISP and C-Prolog.

5.2. Concluding statements

I hope that IPL has illustrated that it is quite feasible to combine Lisp and Prolog programming paradigms by implementing a simple and useful interface. I feel that IPL, as it is currently implemented, can be a useful programming environment for small artificial intelligence projects. Serious projects, having greater computational needs, would

require that IPL be re-implemented to improve run-time efficiency.[1]

It is interesting to speculate on the use of IPL in a multi-processor environment. Run-time performance would be enhanced if the environment had separate Lisp and Prolog hardware "engines". Unfortunately, the current IPL design specifies sequential execution, so one engine would always be idle. This problem could be remedied if a new language were created by merging IPL design concepts with concepts from current research in parallel Prolog execution. In a system employing such a language, when a Prolog engine became blocked while awaiting Lisp evaluation results, it could begin a parallel proof of another goal, thus maximizing processor utilization.

In addition to demonstrating an approach to a Lisp/Prolog programming environment, this thesis has satisfied some personal goals: First, I have learned some of the problems encountered when attempting to interface two dissimilar languages. Second, I am more familiar with an underlying implementation for both Lisp and Prolog. Third, I have become more skilled in Lisp, Prolog, and my usage of the Unix operating system. Finally, IPL has (hopefully) been a suitable vehicle for satisfying academic degree requirements.

[1]Perhaps the greatest efficiency gains could be obtained by replacing the C-Prolog and XLISP interpreters with optimizing Prolog and Lisp compilers.

APPENDIX A

Definition of terminology

Prolog syntax is not standardized, and the terminology used to describe it often varies by author. For this reason this appendix gives a brief introduction to the syntax of C-Prolog [Pereira,85]. The terminology used is that of [Clocksin,84], although I have also parenthetically included commonly used alternate terminology.

Everything in a Prolog program is a *term.* A term is a *constant*, *variable*, or *structure.*

A *constant* is either an *atom*, *integer*, or *floating-point number.*

An *atom* is a string of characters containing one or more of the characters {a-z, A-Z, 0-9, _} and must begin with a lower-case letter, or may be a string of sign characters (+, , *, /, etc.), or may be any characters within single quotes. Examples of atoms are

a abc a01_A = --> ** 'A+B %'

Integers are positive or negative whole numbers, without a decimal point. For example:

-9000 -1 0 25 30000

Floating-point numbers are identified by the presence of a decimal point. For example:

-12.725 3.14159 -0.001

Variables begin with an upper-case letter or an underscore. For example:

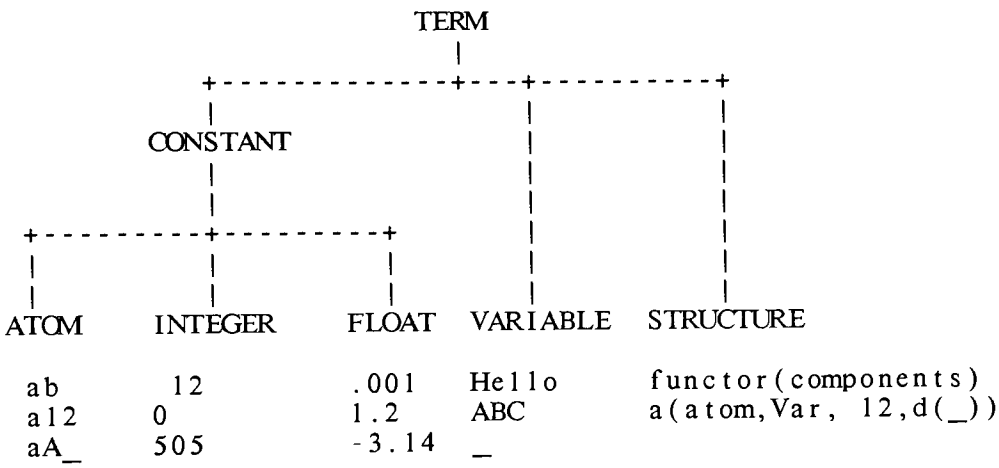
_ _a ABC Hello_there

A variable is initially *uninstantiated*, i.e. it is not bound to any term. Prolog inference may eventually cause a variable to become unified with another term. The variable is then

said to be *instantiated.*

Structures (compound terms) are of the form *functor(components)* where *functor* is an atom and *components* are one or more comma-separated terms. (An alternate terminology for *component* is *argument.*) A functor is said to be of *arity* n, where *n* is the number of components. A shorthand for referring to a functor of arity n is *functor/n.* For example, the built-in predicate **call**, which has one component, is written **call/1**.

The above is perhaps best shown diagrammatically. Examples are shown below each terminal node:



Prolog *lists* are zero or more comma-separated terms within square brackets. For example:

[ab, 0.1, Hi, functor(terms)]

Prolog lists are actually only a notational convenience for ./2 structures, which are structures with the functor "." and having two components. Thus the following structure is identical to the aforementioned list:

.(ab, .(0.1, .(Hi, .(functor(terms), []))))

The C-Prolog equivalent to Lisp "dotted-pairs" is created using the vertical bar "|". For example, **[a|b]** or **[0 1 2 3|4]**. Again, this is only a notational convenience for ./2

structures.

Prolog programs consist of *clauses* of the form

head :- body.

The head or body may be omitted as described below.

The *head* is a structure or atom. If the *body* is present it is one or more structures or atoms separated by commas. Internally this is converted to a single ,/2 structure.

A clause consisting of only a head (a *unit clause*) and containing no variables is often called a *fact*. A clause with both head and body (a *non-unit clause*) is often called a *rule*. A clause consisting of only a body (a *negative clause*) is a *query*.

Unlike Prolog, Lisp syntax and terminology is fairly standardized, thus I will not include those definitions.

APPENDIX B

xprolog operator precedence

(See also [Pereira,85,p.8-9].)

```
:-op(   1200,   fx,     [ :- , ?- ]).
:-op(   1200,   xfx,    [ (:-) , --> ]).
:-op(   1100,   xfy,    ';' ).
:-op(   1050,   xfy,    '->' ).
:-op(   1000,   xfy,    ',' ).
:-op(   950,    fx,     [ : , :: ]).
:-op(   925,    xfx,    ':=' ).
:-op(   920,    fx,     [ ^ , ` , #^ , '.' , '.@' , '"' ]).
:-op(   900,    fy,     [ \+ , not , spy , nospy ]).
:-op(   700,    xfx,    [ = , is , =.. , == , \== ,
                          @< , @> , @=< , @>= ,
                          =:= , =\= , < , > , =< , >= ]).
:-op(   500,    yfx,    [ + ,  , /\ , \/ ]).
:-op(   500,    fx,     [ (+) , (-) , \ ]).
:-op(   400,    yfx,    [ * , / , // , << , >> ]).
:-op(   300,    xfx,    [ mod ]).
:-op(   200,    xfy,    [ (^) ]).
```

APPENDIX C

Lisp and Prolog code

Below are listings of all the Lisp code and most of the Prolog code.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; pxlinit.lsp - Prolog interface functions.
; May 16, 1987          Steven J. Recard        RIT Master's Thesis
;
; MODIFICATION HISTORY
;
; 16May87 sjr - Initial release.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  First a few functions that are used by the Prolog/Lisp
;  interface, but may also be of use to the user.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  (p$setstruct list)
;  Set 'list' to be of Prolog type STRUCTURE.
;  Just a convenient interface to the function 'p$settype'.
;
(defun p$setstruct (list*) (p$settype list* 'structure))


;  (p$structp list)
;  A predicate to test if 'list' is of Prolog type STRUCTURE.
;  If yes, then returns t, else nil.
;
(defun p$structp (list*) (eq (p$type list*) 'structure))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Now some miscellaneous functions used only by the interface.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  ($msq sym)
;  Make String, Quoted.
;  'sym' is not evaluated.  Result is sym's print-name (a string).
;  Used by the Prolog calling interface.
;
(defmacro $msq (sym*) (symbol-name sym*))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Miscellaneous 'private' functions.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  ($princt &rest arglist)
;  For each argument in arglist, 'princ' and 'terpri'.
;
(defun $princt (&rest arglist)
    (dolist (eacharg arglist) (princ eacharg) (terpri)))
```

```
51
52
53      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
54      ;
55      ;  FUNCTIONS THAT DIRECTLY HANDLE THE INTERFACE TO PROLOG.
56      ;
57      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
58
59
60      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
61      ; GLOBAL SYMBOLS USED:
62      ;
63      ; $fromPrologRaw        ; The raw, un-processed "Lisp query".  Exactly as
64      ;                       ; was extracted from the packet.
65      ;                       ; It has not yet been processed --
66      ;                       ; structures and strings have not yet been created.
67      ; $fromProlog           ; Holds the s-exp resulting from evaluation of
68      ;                       ; $fromPrologRaw.  In this sexp, structures and
69      ;                       ; strings have been created.
70      ; $fromPrologEvaled     ; Holds the evaluated result of '$fromProlog'.
71      ;                       ; This is the 'answer' that will be converted
72      ;                       ; to Prolog syntax and shipped back to Prolog.
73      ; $iplTraceFlag         ; Holds the Lisp trace flag sent from Prolog.
74      ; $iplReturnFlag        ; Holds the return flag sent from Prolog.
75      ;                       ; nil means to only return nil; t means to return
76      ;                       ; the evaluated result.
77      ; $fromPrologError      ; Holds the 'Prolog error flag' sent from Prolog.
78      ;                       ; It is also set by Lisp code in certain cases of
79      ;                       ; serious Prolog/Lisp interface errors.
80      ; $lpErrorCount         ; Holds a numeric value indicating the number of
81      ;                       ; errors detected during a given Lisp-to-Prolog
82      ;                       ; conversion.  Used to build "error symbols"
83      ;                       ; (see $lpNewErrorSym) and to set the outgoing
84      ;                       ; packetized error flag.
85      ; $tmp                  ; A scratch variable used in function $rep.
86      ;                       ; Originally $rep declared an aux variable (&aux tmp),
87      ;                       ; but this had undesirable effects when xprolog users
88      ;                       ; defun'ed functions.  (The aux variable appeared at
89      ;                       ; the end of their function definition.)
90      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
91
92
93      ;  Prolog/Lisp interface Read-Eval-Print loop.
94      ;  Uses the global symbols described above.
95      ;  End of file (EOF) processing is necessary to prevent an error from being
96      ;  reported when the user exits Prolog causing Prolog to close the pipes to
97      ;  Lisp.
98      ;  The received packet is of the format:
99      ;       ( <raw Lisp query> <Trace flag> <Return flag> <Prolog error flag> )
100     ;  The returned packet is in the form:
```

```
;       [ <result of Lisp evaluation>, <error flag> ]
;  <error flag> will be true if either <Prolog error flag> was set or if
;  Lisp detects an error during Lisp-to-Prolog conversion (in which case
;  $lpErrorCount > 0).
;
(defun $rep ()
  (prog ()
    outerloop
    (errset
      (prog ()
        (setq *breakenable* nil)
        (setq *tracenable* nil)
        innerloop
        (if (null (setq $tmp (read *prolog-input*))) (exit)) ; EOF test.
        ($fromPrologParse $tmp)                ; Setup the global symbols.
        (if (member $iplTraceFlag '(all receiveraw))
            ($lisptrace 'receiveraw $fromPrologRaw))
        (setq $fromProlog nil)          ; Let's do this just in case ...
        (setq $fromProlog (eval $fromPrologRaw))  ; this line fails.
        (if (member $iplTraceFlag '(all both receive))
            ($lisptrace 'receive $fromProlog))
        (cond
          ($fromPrologError             ; If some error already found, skip
            ($lp $fromProlog t))        ; the evaluation step and echo
                                        ; back to Prolog what was received.
          (t                            ; Else no errors yet.
            (setq $fromPrologEvaled (eval $fromProlog))
            (if (member $iplTraceFlag '(all both eval))
                ($lisptrace 'eval $fromPrologEvaled))
            ; Send to Prolog either the result, or nil, depending upon flag.
            ($lp (if $iplReturnFlag $fromPrologEvaled nil) nil)))
        (go innerloop)
      )                                 ; End prog.
      t )                       ; End errset.  Allow printing of error msgs.
    ; If fall out of errset, then must have trapped an error.
    ($princt "Lisp evaluation error trap!  Lisp received:")
    (print $fromProlog)
    ($lp $fromProlog t)         ; Indicate that an error was found.
    (go outerloop)))


; ($FromPrologParse arg)
; Parse the packet received from Prolog.  Check received packet for
; proper format.  Return the received data via the global symbols
;       $fromPrologRaw  = Holds raw "Lisp query" received by Lisp.
;       $iplTraceFlag   = Holds the "Lisp trace" flag as sent by Lisp.
;                         One of (none, all, both, eval, receiveraw, receive).
;       $iplReturnFlag  = Holds the "return flag" sent by Prolog. t or nil.
;       $fromPrologError= Indicates whether an error was detected by
;                         Prolog during sending out to Lisp.
;
```

```
151     ;                          Either 'nil' or 't'.
152     ;
153     (defun $fromPrologParse (arg)
154         (cond
155             ((and (consp arg) (eql 4 (length arg))) ; If packet looks valid...
156                 (setq $fromPrologRaw (car arg))
157                 (setq $iplTraceFlag (cadr arg))
158                 (setq $iplReturnFlag (eq (caddr arg) t)) ; Defensive programming.
159                 (setq $fromPrologError (eq (cadddr arg) t)) ; Defensive programming.
160                 (if (not ($iplTraceFlagAudit $iplTraceFlag))
161                       (progn
162                         (princ "Bad trace flag: ")
163                         (print $iplTraceFlag)
164                         (setq $fromPrologError t)
165                         (setq $iplTraceFlag 'none))))   ; Set to something valid.
166             ( t     ; Else bad packet.
167                 ($princt "Bad packet received from Prolog:" arg)
168                 (setq $fromPrologError t)   ; Tell Prolog about this.
169                 (setq $fromPrologRaw nil)   ; Set to something valid.
170                 (setq $iplReturnFlag t)     ; Set to something valid.
171                 (setq $iplTraceFlag 'none))))
172
173
174     ; ($iplTraceFlagAudit flag)
175     ; Audit the trace flag sent from Prolog.
176     ; Return 't' if good, else nil.
177     ;
178     (defun $iplTraceFlagAudit (flag)
179         (and
180             (eq (type-of flag) ':symbol)
181             (member flag '(none all both eval receiveraw receive))))
182
183
184     ; ($lisptrace type data)
185     ; Perform a Lisp trace, if possible using the user defined function lisptrace.
186     ;
187     (defun $lisptrace (type data)
188         (cond
189             ((boundp 'lisptrace)
190                 (if (null (errset (lisptrace type data) t))
191                     ($princt "Bad lisptrace function!")))
192             (t  (princ "-- Lisp trace -- <")
193                 (princ type)
194                 ($princt "):")
195                 (print data))))
196
197
198     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
199     ;
200     ;  NOW THE LOWER LEVEL LISP  TO -  PROLOG CONVERSION FUNCTIONS
```

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; ($lp arg err-flag)
; Lisp-to-Prolog, convert to Prolog syntax and send to Prolog.
; 'arg' is evaluated, converted to Prolog, then packetized and sent to Prolog.
; The packet format (in Prolog notation) is:
;       [ <result of Lisp evaluation>, <error flag> ]
; where <error flag> is [] or 't' for error not found or found,
; respectively.  <error flag> is 't' if either err-flag is true or if
; an error was found during Lisp-to-Prolog conversion (i.e., $lpErrorCount>0).
;
(defun $lp (arg err-flag)
    (setq $lpErrorCount 0)      ; Init. to no L-to-P conversion errors found.
    ($lpPrinc "[")
    ($lpInternal arg)
    ($lpPrinc ", ")
    ($lpInternal (or err-flag (plusp $lpErrorCount)))
    ($lpPrinc "]")
    ($lpFullStop))              ; Output necessary Prolog 'full stop'.


; ($lpFullStop)
; Output the necessary Prolog 'full stop'.
;
(defun $lpFullStop ()
    ($lpPrinc ".")
    ($lpTerpri)
    ($lpFlusho))


; ($lpInternal arg)
; Lisp-to-Prolog $lp internal routine.
; A private routine for use by the function '$lp'.
;
(defun $lpInternal (arg)
    (case (type-of arg)
        ( (nil)
            ($lpPrincNil))
        ( (:string)
            ($lpPrincString arg))
        ( (:fixnum :flonum)
            ($lpPrincNumber arg))
        ( (:symbol)
            ($lpPrincSym arg))
        ( (:cons)
            (if (p$structp arg)
                ($lpPrincStruct arg)
                ($lpPrincList arg)))
        ( (:object :fsubr :subr :file :array)
```

```
251                 ($princt "Lisp-to-Prolog conversion error.  Unsupported data type.")
252                 (princ "Can't pass it to Prolog:  ")
253                 (print arg)
254                 (princ "Symbol ")
255                 (princ ($lpPrincErrorSym))
256                 ($princt " substituted.")))
257
258
259       ; ($lpPrincNil)
260       ; "Lisp-to-Prolog, print a NIL.
261       ;
262       (defun $lpPrincNil () ($lpPrinc "[]"))
263
264
265       ; ($lpPrincNumber num)
266       ; "Lisp-to-Prolog, print a fixed or floating point number.
267       ;
268       (defun $lpPrincNumber (num)
269           ($lpPrinc num))
270
271
272       ; ($lpPrincSym str)
273       ; Lisp-to-Prolog, print a symbol.  (I.e., a Prolog "atom".)
274       ;  $pprinc correctly handles symbols containing single quotes -- two quotes
275       ;  are output.  For example, ($pprinc '|ab'cd|) will output
276       ;       'ab''cd'
277       ;
278       (defun $lpPrincSym (sym)
279           ($pprinc sym *prolog-output*)
280           sym)
281
282
283       ; ($lpPrincString str)
284       ; Lisp-to-Prolog, print a string.
285       ;
286       (defun $lpPrincString (str)
287           ($lpPrincSym "\"")
288           ($lpPrinc "(")
289           ($lpPrincSym str)
290           ($lpPrinc ")"))
291
292
293       ; ($lpPrinc atom)
294       ; Lisp-to-Prolog, print 'atom' without newline terminator.
295       ;
296       (defmacro $lpPrinc (atom) `(princ ,atom *prolog-output*))
297
298
299       ; ($lpTerpri)
300       ; Lisp-to-Prolog, terminate the line (i.e., output a newline).
```

```
301     ;
302     (defmacro $lpTerpri () (terpri *prolog-output*))
303
304
305     ; ($lpFlusho)
306     ; Lisp-to-Prolog, flush the output buffers.
307     ; I.e., write out any output buffers.
308     (defmacro $lpFlusho () ($flusho *prolog-output*))
309
310
311     ; ($lpPrincLisp ListArg)
312     ; List-to-Prolog, print a list in Prolog syntax.
313     ; Private routine for $lpInternal.
314     ;
315     ; Note: The catch and throw are necessary only for the case when the
316     ;       final element of the "dotted-pair" is a structure.  In this case,
317     ;       in Lisp it really isn't a dotted pair, but will be when converted
318     ;       into Prolog.  For example, consider the following list
319     ;               (a b f c d e)
320     ;       in which the list beginning at 'f' is marked as 'structure'.
321     ;       The Prolog equivalent to this list is
322     ;               [a, b | f(c, d, e)]
323     ;       The catch and throw are used to prevent the 'mapl' function from
324     ;       running through the list after element 'f'.
325     ;
326     ;       As an aside, the Prolog
327     ;               [a, b, f(c, d, e)]
328     ;       maps to the Lisp
329     ;               (a b (f c d e))
330     ;
331     (defun $lpPrincList (ListArg)
332         ($lpPrinc "[")
333         (catch 'at_end_of_dotted_pair
334             (mapl
335                 #'(lambda (arg)
336                     ($lpInternal (car arg))
337                     (cond
338                         ( (null (cdr arg))) ; If at end of list, do nothing.
339                         ( (and              ; If more list (but not structure)
340                             (consp (cdr arg))       ;   to come...
341                             (not (p$structp (cdr arg))))
342                           ($lpPrinc ",")    ; Print comma separator.
343                           ($lpTerpri)
344                         )
345                         ( t                 ; Else arg must be a dotted-pair.
346                           ($lpPrinc " | ")
347                           ($lpInternal (cdr arg))
348                           (throw 'at_end_of_dotted_pair))))
349                 ListArg)
350         )   ; end of catch.
```

```
351           ($lpPrinc "]"))
352
353
354       ; ($lpPrincStruct struct)
355       ; List-to-Prolog, print a structure in Prolog syntax.
356       ; Private routine for $lpInternal.
357       ;
358       (defun $lpPrincStruct (struct)
359         (cond
360           (((< (length struct) 2)
361             ($princt "Lisp-to-Prolog conversion error.  Bad Structure:")
362             (print struct)
363             (princ "Structure must be of at least arity 1.  Symbol ")
364             (princ ($lpPrincErrorSym))
365             ($princt " substituted."))
366           ; Note: The above will also catch dotted pairs marked as structures,
367           ;   e.g. (p$setstruct '(a . b))
368           ((not (eq (type-of (car struct)) ':symbol))
369             ($princt "Lisp-to-Prolog conversion error.  Bad structure:")
370             (print struct)
371             ($princt "Structure must have valid Prolog atom as functor!")
372             (princ "Symbol ")
373             (princ ($lpPrincErrorSym))
374             ($princt " substituted."))
375           ( t                          ; Looks like good structure so far.
376             ($lpPrincSym (car struct))        ; Print the functor.
377             ($lpPrinc "(")                    ; Print the opening parens.
378               (mapl
379                 #'(lambda (arg)
380                     ($lpInternal (car arg))   ; Process the next argument..
381                              ; Now look ahead to determine the delimiter.
382                     (if (not (null (cdr arg))) ; If more arguments...
383                         ($lpPrinc ","))       ; Print the argument delimiter.
384                         ($lpTerpri)
385                     (if (not (listp (cdr arg)))  ; If (cdr arg) is a non-nil atom,
386                                                  ; then we have a dotted-pair within a
387                                                  ; structure.  Error!
388                       (progn
389                         (princ "Lisp-to-Prolog conversion error.  ")
390                         ($princt "Bad Structure:")
391                         (print struct)
392                         (princ "Structure arguments cannot be dotted pairs!")
393                         (princ "  Symbol ")
394                         (princ ($lpPrincErrorSym))
395                         ($princt " substituted.")))
396                   )                          ; End lambda.
397               (cdr struct))   ; End mapl.  Process the structure arguments.
398             ($lpPrinc ")"))))         ; Print the closing structure parens.
399
400
```

```
401     ; $lpNewErrorSym -- Generate an 'error symbol' for shipping to Prolog.
402     ; Uses the global symbol '$lpErrorCount'; INCREMENTS IT AFTER USE.
403     ; If the optional parameter 'str' is omitted, the error symbol
404     ; will be of the format
405     ;       >>err-N-err<<   , where N is some integer.
406     ; If 'str' is specified, the symbol will be of the form
407     ;       >>STR-N-STR<<          , where N is some integer and STR is 'str'.
408     ; The integer N is obtained from '$lpErrorCount'.
409     ;
410     (defun $lpNewErrorSym (&optional str &aux newsym &aux stream)
411         (setq stream (cons nil nil))        ; Create a stream.
412         (if (null str) (setq str 'err))     ; Use default string if necessary.
413         (princ ">>" stream)
414         (princ str stream)
415         (princ "-" stream)
416         (princ $lpErrorCount stream)
417         (princ "-" stream)
418         (princ str stream)
419         (princ "<<" stream)
420         (terpri stream)
421         (setq newsym (read-line stream))
422         (setq $lpErrorCount (1+ $lpErrorCount))
423         newsym)
424
425
426     ; ($lpPrincErrorSym &optional str)
427     ; Output to Prolog the next error symbol.  Returns the error symbol.
428     ;
429     (defun $lpPrincErrorSym (&optional str &aux newsym)
430         (setq newsym ($lpNewErrorSym str))
431         ($lpPrinc " ")
432         ($lpPrincSym newsym)
433         ($lpPrinc " ")
434         newsym)
435
436
437     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
438     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
439     ;
440     ;  Now initialize XLISP to communicate with Prolog via pipes.
441     ;
442     ;  When Prolog forks off XLISP, it redirects stdin and stdout to
443     ;  the pipes.  All other file descriptors are closed, except stderr.
444     ;
445     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
446     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
447
448     (setq *prolog-input* *standard-input*)
449     (setq *prolog-output* *standard-output*)
450     (setq *standard-input* (openi "/dev/tty"))
```

```
451       (setq *standard-output* (openo "/dev/tty"))
452       ($unbuf *standard-output*)      ; Necessary for Pyramid O.S.
453                                       ; Not necessary for AT&T Unix PC version 3.0.
454
455
456       ($rep)          ; Begin Lisp/Prolog interface Read-Eval-Print loop.
457
458       ; end-of-file.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% lisp -- Rules for Prolog to XLISP interface.
% May 6, 1987          Steven J. Recard        RIT Master's Thesis
%
% MODIFICATION HISTORY:
%
% 06May87 sjr - Initial attempt.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% NOTES:
%
%  1.   Data sent to Lisp is packetized into the following Lisp list:
%       ( <stuff to be eval'ed by Lisp> <Lisp trace flag> <return data flag>
%         <error flag> )
%
%       <stuff to be eval'ed by Lisp> must be eval'd by Lisp TWICE to get
%       the final result.  The first evaluation creates the necessary
%       structures and strings and results in the s-exp to be finally
%       evaluated.  The second evaluation generates the actual answer
%       that is shipped back to Prolog.
%
%       <Lisp trace flag> is set to one of
%              [all, none, both, eval, receiveraw, receive].
%       See lisptrace/1 for more info.
%
%       <return data flag> indicates whether the result of final evaluation
%       is to be returned back to Prolog.  Either 't' or ().
%       If (), then the returned data is always [].  (Unless an error
%       condition is found, in which case Lisp will 'echo back' what it
%       received.  See below.)
%       The Prolog predicate '::' sets this flag to ().  (Otherwise
%       statements such as    ::[setq, fp, [openo, '" 'sjr.tmp']]
%       would fail because setq would return a file pointer.)
%       The other Lisp call predicates set it to 't'.
%
%       <error flag> is set to () if no error found, else 't'.
%
%       Data received from Lisp is received packetized as follows:
%       [ <result of Lisp evaluation>, <error flag> ]
%
%       <error flag> is [] if error found, else 't'.  <error flag> will
%       always be set true if the outgoing packet error flag was true.
%
%  2.   Error handling strategy for errors detected while sending to Lisp:
%
%       The basic philosophy is "always send something syntatically valid".
%       The problem is that once data has been sent to Lisp,
```

```
51      %       it can't be retrieved.  Even though the output data is buffered,
52      %       enough data may have already been sent such that some of the output
53      %       may already have been read by Lisp.  So...
54      %
55      %       I always send something that is syntactically valid,
56      %       if necessary substituting something valid for the invalid data.
57      %       The <error flag> is then set in the outgoing packet to indicate
58      %       that an error was found.  Lisp will test the error flag,
59      %       and if set, Lisp won't eval the received data, but rather will
60      %       just 'echo' it back.
61      %
62      %       The alternative would have been to write some C code to signal
63      %       Lisp to flush its input buffers and "re-sync".  This
64      %       approach was not followed since it would have required
65      %       implementation via C/Unix, and thus was messier than the
66      %       Lisp/Prolog language approach that I employed.  (I wanted to
67      %       stay as "hi-level" as possible.)
68      %
69      %       The Lisp to Prolog error handling is performed similarly.
70      %
71      %  3.   The structure $lispTraceMode/1 is employed to indicate whether
72      %       Lisp tracing is set.  See the evaluable predicate lisptrace/1 for
73      %       more info.
74      %
75      %  4.   The atom $plConversionErrorFound is used to remember whether or not
76      %       an error was encountered while sending out the data to Lisp.
77      %       If this atom is asserted, then an error was found and the
78      %       error flag will be asserted in the outgoing packet to Lisp.
79      %
80      % BUGS:
81      %  1.   The '#unbound#' problem:  if the symbol '#unbound#' is ever
82      %       passed to Lisp, it will cause problems in the Lisp interface
83      %       code.  For this reason, Prolog maps '#unbound#' to '##unbound##'.
84      %       It is kludgy, but at least it prevents the Lisp interface from
85      %       blowing up.
86      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
87
88
89      %-----------------------------------------------------------------------
90      % Send 'To_Lisp' to Lisp, try to unify result with 'Result'.
91      % The 't' parameter means to return result.
92      %-----------------------------------------------------------------------
93      :=(Result, To_Lisp) :-
94              $lispcall(LispResult, To_Lisp, t), !, Result = LispResult.
95
96
97      %-----------------------------------------------------------------------
98      %       Internal routine to make a call to Lisp and return the returned
99      %       data via 'Result'.  'Return_flag' must be 't' or [].
100     %       If 'Return_flag' is [], the returned result will always be [];
```

```
101     %       else returned result will be whatever Lisp evaluated.
102     %-----------------------------------------------------------------------
103     $lispcall(Result, To_Lisp, Return_flag) :-
104             $lispSend(To_Lisp, Return_flag), !, $lispReceive(Result).
105
106
107     $lispSend(X, Return_flag) :-
108                             % Note: The design assumes that $plOut will always
109                             %       succeed.  If it does fail it is catastrophic
110                             %       and the next $lispSend clause will be forced
111                             %       to cleanup.
112         telling(Save),
113         tell($lispio),
114         $lispSendIt(X, Return_flag),        % Send X to Lisp.
115         tell(Save).
116
117     $lispSend(_,_) :-               % Error handling for just in case...
118         display('Fatal $lispSend error!'), ttynl, $lispFatalInterfaceError.
119
120
121     $lispSendIt(X, Return_flag) :-  % Convert X into Lisp and send it.
122         abolish($plConversionErrorFound,0),         % Set to "no error found".
123         write('('), $plOut(X),
124         lisptrace(Trace), $plOut(Trace), $plOutDelim,
125         $plOut(Return_flag), $plOutDelim,
126         $getErrFlag(ErrFlag), $plOut(ErrFlag), write(')'), nl, $flush.
127
128
129     $getErrFlag(ErrFlag) :- % Check if Prolog-to-Lisp conversion error occurred.
130         $plConversionErrorFound, !, ErrFlag = t.
131
132     $getErrFlag([]).        % If no conversion error, then return nil.
133
134
135     $lispReceive(Result) :-                 % Receive from Lisp.
136         seeing(Save), see($lispio), $tryRead(In, Save), !,
137         $lispReceiveParse(In, Result).
138
139     $tryRead(In, Save) :- read(In), see(Save).
140
141     $tryRead(In, Save) :- see(Save), !, fail.       % Cleanup after read error.
142
143
144     $lispReceiveParse([Result | [ErrFlag]], Result) :- !,
145         $iplErrorProcess(ErrFlag, Result).
146
147     $lispReceiveParse(Input, _) :-  % This only used if bad packet received.
148         display('Improper packet received from Lisp:'), ttynl,
149         display(Input), ttynl, $lispFatalInterfaceError.
150
```

```
151
152     $iplErrorProcess([], _) :- !.   % If Lisp returned nil for error flag,
153                                     % then no error.
154
155     $iplErrorProcess(t, Result) :-  % If error was found, then display what
156                                     % Lisp sent back.
157         display('Prolog form:'), ttynl, display(Result),
158         ttynl, !, fail.
159
160     $iplErrorProcess(BadFlag,_) :-  % Bad packet received.
161         display('Bad packet received -- garbage error flag: '),
162         display(BadFlag), ttynl, $lispFatalInterfaceError.
163
164
165     $lispFatalInterfaceError :-
166         display('Fatal Prolog/Lisp interface error!  Lisp being aborted!'),
167         ttynl, close($lispio), abort.
168
169
170     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
171     %  Now all the clauses for conversion and output of Prolog terms to Lisp.
172     %  The quoting of 'comma' and 'comma-at' is actually only necessary
173     %  when those symbols occur within the context of a list, but
174     %  (a) it doesn't hurt to always quote them and
175     %  (b) doing the check here results in the cleanest program structure.
176     %  The quoting is necessary because the Prolog/Lisp interface
177     %  (on the Lisp side) uses Lisp backquoting for its own processing.
178     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
179
180     $plOut(Term) :-         % Handle uninstantiated variables.
181         var(Term), !, $plAtomOut(Term),
182         display('Error: Can''t pass uninstantiated variables to Lisp!'),
183         display('  Symbol '), display(Term), display(' sent.'), ttynl,
184         assert($plConversionErrorFound).
185
186     $plOut([]) :- !, $plNilOut.
187
188     $plOut(comma) :- !, write(',''comma').
189
190     $plOut('comma-at') :- !, write(',''comma-at').
191
192     $plOut('*unbound*') :- !, write('**unbound**').         % Kludge for a Lisp bug.
193
194     $plOut(Term) :- atom(Term), !, $plAtomOut(Term).
195
196     $plOut(Term) :- number(Term), !, $plNumberOut(Term).
197
198     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
199     %  The following clauses handle the structures having special
200     %  meaning to the Prolog-to-Lisp interface.
```

```
201     %  All of these clauses are defined as prefix operators.
202     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
203
204     $plOut('"'(Term)) :- atom(Term), Term \== [], !, $plStringOut(Term).
205
206     $plOut('"'(BadTerm)) :- !,
207         display('Error: String is not an atom: '), display(BadTerm), ttynl,
208         display('String >>bad-string<< substituted.'), ttynl,
209         assert($plConversionErrorFound),
210         $plOut('"'('>>bad-string<<')).
211
212     $plOut('^'(Term)) :- !, $plOut([quote, Term]).
213
214     $plOut('#^'(Term)) :- !, $plOut([function, Term]).
215
216     $plOut('`'(Term)) :- !, $plOut([backquote, Term]).
217
218     $plOut(','(Term)) :- !, $plOut([comma, Term]).
219
220     $plOut(',@'(Term)) :- !, $plOut(['comma-at', Term]).
221
222     %
223     %  End of "special" clauses.
224     %
225
226     $plOut(Term) :- Term = [_|_], !, $plListOut(Term).
227
228     $plOut(Term) :- Term =.. _, !, $plStructOut(Term).
229
230     $plOut(Mystery) :-      % Error!
231         !, $plNilOut,
232         display('Aarrghh! What are you trying to pass to Lisp? < '),
233         display(Mystery), display(' >'), ttynl, display('nil sent instead!'),
234         ttynl, assert($plConversionErrorFound).
235
236
237     $plNilOut :- write( '()' ).
238
239             % The following clause is also used to output uninstantiated variables.
240             % $writeIq/1 will output the atom 'ab!c\d' as  !ab\!c\\d!
241     $plAtomOut(Atom) :-  $writeIq(Atom).
242
243             % Concerning $plNumberOut/1: I don't check for numbers beyond the range
244             % acceptable to XLISP.  If the number is too large Prolog will
245             % convert it to scientific notation, and XLISP 1.6 will treat it as a
246             % symbol!  XLISP 1.6 doesn't accept scientific notation.
247     $plNumberOut(Num) :- write(Num).
248
249     $plStringOut(Str) :- write( ',($msq ' ), $plAtomOut(Str), write( ')' ).
250
```

```
251
252     %%%
253     %  $plListOut*/1 clauses:
254     %  These clauses format a list and output it to Lisp.
255     %%%
256
257     $plListOut(List) :-     !, write( '(' ), $plListOutInside(List), write( ')' ).
258
259     $plListOutInside([]) :- !.      % If at end of list, then all done.
260     $plListOutInside([H|T]) :- !,   % Else recurse through the list.
261         $plOut(H), $plListElementDelimiterOut(T), $plListOutInside(T).
262
263                     % The following rule only used for right side of a dotted pair.
264     $plListOutInside(X) :-  $plOut(X).
265
266
267     %%%
268     % Convert Prolog structures to Lisp and output.
269     %%%
270
271     $plStructOut(Struct) :-
272         write( '(p$setstruct '' ), Struct =.. List, $plListOut(List),
273         write( ')' ).
274
275
276     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
277     %  Now the clauses to output the delimiter (space or dot) between terms
278     %  inside of lists or structures.
279     %
280     %  Comments for the next three clauses, in clause order:
281     %  1.   If at end of list, don't bother putting out delimiter.
282     %  2.   If the remainder of the list is a list, then just put out
283     %       whitespace for the delimiter.
284     %  3.   If the remainder of the list isn't nil or another list, then we must
285     %       have a 'dotted-pair'.  Put out appropriate Lisp delimiter.
286     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
287
288     $plListElementDelimiterOut([]) :- !.                    % comment #1.
289     $plListElementDelimiterOut([H|T]) :- $plOutDelim, !.    % comment #2.
290     $plListElementDelimiterOut(_) :- write( ' . ' ).        % comment #3.
291
292     $plOutDelim :- nl.      % Output delimiters between symbols.
293                             % Use newline to prevent Lisp input buffer overflow.
294
295     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
296     %  Miscellaneous other useful evaluable predicates available to the user.
297     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
298
299                     % Same as ':=' but Lisp result is displayed to tty.
300                     % Defined as prefix operator.
```

```
':'(ToLisp)  :- Result := ToLisp, display(Result), ttynl.


                % Same as ':=' but Lisp result is thrown out.
                % Defined as prefix operator.
'::'(ToLisp) :- $lispcall(_, ToLisp, []).      % [] means to return only [].


                % Same as ':=', but result is tested for inequality
                % against [].  Succeeds if result of Lisp is not [], else fails.
lisptest(ToLisp) :-  Result := ToLisp, !, Result \== [].


golisp :-       % Allow the user to switch to Lisp read-eval-print loop.
    display('Execute ''(continue)'' to return to Prolog.'), ttynl,
    ::[break, '"'('Entering Lisp.')].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Now the Lisp tracing clauses.
%  The structure $lispTraceMode/1 is used to remember the current
%  tracing mode.  If it doesn't exist in the database, then mode
%  'none' is assumed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lisptrace(X) :-         var(X), !, $lispTraceCurrentMode(X).
lisptrace(none) :-      !, abolish($lispTraceMode,1).
lisptrace(receiveraw) :- $lispTraceSet(receiveraw).
lisptrace(receive) :-   $lispTraceSet(receive).
lisptrace(eval) :-      $lispTraceSet(eval).
lisptrace(both) :-      $lispTraceSet(both).
lisptrace(all) :-       $lispTraceSet(all).
lisptrace(X) :-         display('! Invalid lisptrace specification : '),
                        display(X), nl, !, fail.

$lispTraceCurrentMode(X) :- $lispTraceMode(X), !.
$lispTraceCurrentMode(none).

$lispTraceSet(X) :-     !, abolish($lispTraceMode,1), assert($lispTraceMode(X)).

% end-of-file.
```

APPENDIX D

IPL User's Manual

The following is a copy of the separate document "IPL User's Manual".

IPL User's Manual

Steven J. Recard

Rochester Institute of Technology
School of Computer Science and Technology

June 20, 1987

TABLE OF CONTENTS

1. Introduction

IPL, Interfaced Prolog/Lisp, was the result of a Master's Thesis by Steven J. Recard. This user's manual will provide a brief summary of the IPL system. A complete description can be found in his thesis report [Recard,87]. IPL is the name given to a programming system that was created by making additions and modifications to the C-Prolog 1.4 interpreter [Pereira,85] and to the XLISP 1.6 interpreter [Betz,86]. This document should be used as a supplement to the manuals accompanying these two languages.

The modified Prolog interpreter is called *xprolog*. The modified Lisp interpreter is called *pxlisp*. xprolog has the same definition as C-Prolog except for the additions necessary for IPL. The same is true of pxlisp with respect to XLISP except for two modifications. First, pxlisp supports the vertical bar notation for symbol delimiters. For example, |**Very strange\|symbol\\here**| can be used to enter the symbol **Very strange|symbol\here**. Second, pxlisp was changed to be alphabetic case-sensitive. Symbols are no longer automatically converted to upper-case upon input.

2. Data conversions

The following are the data conversion rules for data passed between xprolog and pxlisp.

(1) Atoms map to atoms without change. For example:

Prolog	Lisp
abc	abc
'A ;'	|A ;|
[]	nil
t	t

(Note: the vertical bars in |A ;| represent symbol delimiters. Similarly, the single quotes in 'A ;' are symbol delimiters in Prolog used to specify an atom having spe-

cial characters.)

(2) Integers map to integers without change. For example:

Prolog	Lisp
0	0
1	1
137	137
-5	-5

(3) Floating point numbers map to floating point numbers without change. For example:

Prolog	Lisp
3.14	3.14
.017	.017

(4) Prolog structures map to Lisp lists with

(a) the Prolog functor as the head of the list and the structure's components as the remaining elements of the list, and

(b) the Lisp list tagged internally as "Prolog type = structure". Tags are associated with the node, or "cons cell",

which starts the list.

For example (internal type tagging not shown):

Prolog	Lisp
a(b,c)	(a b c)
+(a,b)	(+ a b)
a(b,c(d,e),f)	(a b (c d e) f)

(5) Prolog lists map directly to Lisp lists, but with the Lisp list tagged internally as "Prolog type = list".

For example (internal type tagging not shown):

Prolog	Lisp
[1, 2, a]	(1 2 a)
[car, [quote, [a, b]]]	(car (quote (a b)))
[a | b]	(a . b)
[a, b, c | d]	(a b c . d)

(6) Lisp strings map to Prolog structures with functors " (double-quote). For convenience, " is defined as a prefix operator. For example:

Prolog	Lisp
'"' abc	"abc"
'"' 'A,B'	"A,B"

Unsupported data types

The following Lisp (XLISP 1.6) data types are not supported by IPL: objects (for object-oriented programming), arrays, file pointers, subrs (built-in functions), and fsubrs (special built-in functions).

The only restriction placed on Prolog data that will be Lisp evaluated is that all variables must be instantiated.

Attempts to pass data of unsupported type will cause a run-time error message to be printed and the Lisp call predicate to fail. For example, the following statements will cause an error:

```
?- A := car.  /* Cannot return the built-in function "car". */
?- A := '*standard-input*'.  /* Cannot return a file pointer. */
?- :[print, ^A].  /* "A" is uninstantiated, thus illegal. */
```

3. xprolog features

3.1. Lisp call predicates

xprolog supports four Lisp call predicates. All take a Prolog term, convert it to Lisp and allow Lisp to evaluate it. The handling of the result of evaluation depends upon the particular predicate. A Lisp evaluation error will result in an error message being displayed and goal failure. None of these predicates will be re-satisfied on backtracking.

Result := *LispData*

LispData is evaluated by Lisp and the result is unified with *Result*.

: *LispData*

LispData is evaluated by Lisp. The result is written to the user's console via the **write**/1 predicate. : is defined as

:(X) :- Result := X, write(Result), nl.

This predicate is primarily intended for user interactive work with Lisp. It always succeeds, unless there is an error.

:: *LispData*

LispData is evaluated by Lisp. The result is discarded. This predicate is useful for evaluating code whose output is a side-effect. Conceptually, its definition is

::(X) :- _ := X.

It always succeeds, unless there is an error.

lisptest(*LispData***)**

LispData is evaluated by Lisp and the result is tested against []. The goal succeeds if the result is not equal to [], else it fails. It is defined as

lisptest(X) :- Result := X, !, Result \== [].

3.2. Debugging and miscellaneous predicates

golisp

If evaluated, this predicate will suspend the Prolog interpreter and will switch the user to the Lisp interpreter. The user may then, for example, debug Lisp functions or may manually initialize the Lisp environment (by loading files, etc). The user should execute **(continue)** to return to Prolog.

While switched to Lisp the user will be at a break loop prompt with breaks disabled. If desired, the user may re-enable breaks by setting ***breakenable*** to t, but it should be set back to **nil** before returning to Prolog. If it is not reset, Lisp evaluation errors will automatically put the user into the Lisp break loop.

lisptrace(*Mode***)**

This predicate is used for debugging IPL programs. It controls the tracing of Lisp call data. The single component must be one of the following six atoms: **none**, **receive**, **eval**, **both**, **receiveraw**, or **all**. If the component is an uninstantiated variable, the variable will be unified with the current **lisptrace** setting.

The six atoms have the following meanings:

none

Display nothing, i.e. turn off Lisp call tracing. This is the initial setting.

receiveraw

Display the "raw" data as received from Prolog. This is used for debugging the IPL interface itself and is not intended for use by the typical user.

receive

Display the data sent from Prolog that is to be Lisp evaluated.

eval

Display the result of Lisp evaluation.

both

Both the **receive** and **eval** data are displayed.

all

receiveraw, **receive**, and **eval** data are all displayed. This is probably only of interest to IPL "guru-types".

The data is printed to the user's console by the Lisp interpreter using the **print** function. Note that internal list/structure tags are not shown by **print**. A hook exists, however, for the user to supply an alternate output function. Before the data is printed the interface software checks for the existence of a Lisp function named **lisptrace**. If it exists it will be called with two arguments. The first will be either the (quoted) symbol **receiveraw**, **receive**, or **eval**. The second argument will be the trace data. This function could be used to display the trace data in such a way that the list/structure tags are apparent, or to write the trace data to a file for later analysis.

3.3. Representation of Lisp strings in Prolog

The unary prefix operator " is employed to represent Lisp strings in Prolog. Thus '"' **atom** (or the equivalent '"'(**atom**)) is the Prolog form of the Lisp string "**atom**".

3.4. Prolog substitutes for Lisp read-macros

Five unary prefix operators were defined to substitute for commonly used Lisp read-macros:

Lisp syntax	XLISP function equivalent	Prolog syntax
'expr	(quote expr)	^expr
#'expr	(function expr)	#^expr
`expr	(backquote expr)	`expr
,expr	(comma expr)	','expr
,@expr	(comma-at expr)	',@'expr

While these are called unary prefix "operators", they are not Prolog evaluable — they only have meaning to the IPL interface software. In addition, while they are

defined as unary prefix operators, they can also be entered using structure notation. For example, **^(expr)** is also equivalent to **^expr**. The former may be preferable in certain cases to avoid problems of operator precedence.

Since these operators are Prolog versions of "read-macros", the following will not be true:

```
?- ^a := ^ ^a.
```

Lisp will return **[quote, a]**, not **^a**.

3.5. Prolog parser caveat

The user must be cognizant of Prolog syntax rules when using any of the new Prolog operators. White-space or structure notation is sometimes necessary for correct Prolog interpretation. A few examples will illustrate this:

```
: ^ hello. /* Works OK. Note white-space after : operator. */
:'^'hello. /* This is OK. */
: ^(hello). /* This is OK. */
:^ hello. /* Error —:^ taken as single atom. */
:'"' 'Hello, world'. /* OK. Note white-space after '"' */
:'"''Hello, world'. /* Error: Lisp atom will be |"'Hello, world| */
```

3.6. xprolog operator precedence

(See also [Pereira,85,p.8-9].)

```
:-op(   1200,   fx,     [ :- , ?- ]).
:-op(   1200,   xfx,    [ (:-) , --> ]).
:-op(   1100,   xfy,    ';' ).
:-op(   1050,   xfy,    '->' ).
:-op(   1000,   xfy,    ',' ).
:-op(   950,    fx,     [ : , :: ]).
```

```
:-op(    925,    xfx,    ':=' ).
:-op(    920,    fx,     [ ^ , ` , #^ , ',' , ',@' , '"' ]).
:-op(    900,    fy,     [ \+ , not , spy , nospy ]).
:-op(    700,    xfx,    [ = , is , =.. , == , \== ,
                           @< , @> , @=< , @>= ,
                           =:= , =\= , < , > , =< , >= ]).
:-op(    500,    yfx,    [ + , - , /\ , \/ ]).
:-op(    500,    fx,     [ (+) , (-) , \ ]).
:-op(    400,    yfx,    [ * , / , // , << , >> ]).
:-op(    300,    xfx,    [ mod ]).
:-op(    200,    xfy,    [ (^) ]).
```

4. Lisp features

The following functions are defined to set and read the list/structure tags:

(p$settype <list> <tag>)

<list> must evaluate to a list. <tag> must evaluate to either the atom list or **structure**. The first cons cell of <list> will be tagged to be of Prolog type <tag>.

(p$type <list>)

<list> must evaluate to a list. Returns the Prolog type of the list, either **list** or **structure**.

For convenience, two other functions are defined as follows:

```
(defun p$setstruct (arg) (p$settype arg 'structure))
(defun p$structp (arg) (eq (p$type arg) 'structure))
```

(p$setstruct arg) sets **arg** to be of type **structure**. **(p$structp arg)** is a Lisp predicate func-

tion which returns **t** if arg is tagged as structure, else returns **nil**.

Tags are associated with the cons cell starting the list. **nil** is always of Prolog-type **list**.

5. Using the IPL system

IPL is used by executing "xprolog". C-Prolog command line options may be specified, if desired. xprolog will load pxlisp only if needed, specifically only if a Lisp call predicate is evaluated. Once loaded, pxlisp will remain active until either the user exits xprolog or executes the Prolog goal **close($lispio)**.

When pxlisp is automatically loaded it searches the user's current account for a file named "init.lsp". If it finds such a file it will be loaded.

pxlisp can also be used "stand-alone" by executing "pxlisp". The usual XLISP command line options may be specified.

6. Example IPL session

The following is a listing of an actual session using IPL. Comments are shown in *italics.*

```
1 % cat demo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Define a few predicates for purpose of the demo.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

concat(SymBegin, SymEnd, Result) :-
        '"'Result := strcat('"'SymBegin, '"'SymEnd).

sumlist(List, Sum) :- Sum := [apply, #^ +, ^List].

lispprint(X) :- ::[print, ^X].

islist(L) :- lisptest( [listp, [last,^L]] ).

:- op(950, fx, ':^').   % I mistype this alot, so I'll just define it.
:^(X)  :-  :(^(X)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Show how Lisp environment can be conveniently initialized.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- A := [load, '"' 'pp.lsp'],
        ((A == [], display('Load of pp.lsp failed!'), !, fail)
         ;
         true).

% -eof-
2 %
```

The following file is automatically loaded by init.lsp.

```
2 % cat iplinit.lsp
;  IPL user initialization file.                Steven J. Recard
;
; MODIFICATION HISTORY:
; 17May87 sjr - Initial release.

;  (|,| first second)
;  Recursively evaluate  the Prolog comma structure (','/2).
;  This is used to to allow Prolog to pass 'comma structures'
;  to Lisp for evaluation.  It allows a sort of short-hand progn
;  to be used in Prolog.  Returns value of second.
;
(defun |,| (first second) second)

3 % xprolog
C-Prolog version 1.4 ("xprolog")
| ?- [demo].
; loading "pp.lsp"
demo consulted 636 bytes 0.283333 sec.
```

```
yes
| ?- : ^ hello.
hello

yes
| ?- : ^ ^ hello.
[quote,hello]

yes
| ?- A := ^ '[a, ',b, ',@'c, d].

A = [backquote,[a,[comma,b],[comma-at,c],d]]

yes
| ?- A = prolog, B = 'Lisp', concat(A, B, IPL).

A = prolog
B = Lisp
IPL = prologLisp ;

no
| ?- :[setq, x, ^struct('"'string)].
struct("string)

yes
| ?- :x.
struct("string)

yes
| ?- X_value := x, display(X_value).
struct("string)
X_value = struct("string)

yes
| ?- :setq(y, x).
struct("string)

yes
                The following employs the |,| Lisp function defined
                in iplinit.lsp.  It acts the same as the Lisp progn function.
| ?- :: ([princ, y], [princ, '"' ' is of Prolog-type '],
|                 [print, [p$type, y]]).
(struct string) is of Prolog-type structure

yes
| ?- : ^['Hello', func(a,b)].
[Hello,func(a,b)]

yes
| ?- : ^ ['Hello', Var, func(a,b)].
Error: Can't pass uninstantiated variables to Lisp!  Symbol _0 sent.
Prolog form:
[quote,[Hello,_0,func(a,b)]]

no
```

```
| ?- lisptest([p$structp, ^a(b)]).

yes
| ?- lisptest( [p$structp, ^ [a,b]] ).

no
| ?- islist( [a, b] ).

yes
| ?- islist( [a, b | c] ).

no
| ?- name(longsymbolname, L), SymLen := [length, ^L].

L = [108,111,110,103,115,121,109,98,111,108,110,97,109,101]
SymLen = 14

yes
| ?- Ans := 1+2*3.5.

Ans = 8

yes
| ?- L1 = [a,b], L2 = [c,d],
|    L3 := [setq, ans, '[1, ',@' ^ L1, 2, ','[cdr, ^L2], 3]].

L1 = [a,b]
L2 = [c,d]
L3 = [1,a,b,2,[d],3] ;

no
| ?- golisp.
Execute '(continue)' to return to Prolog.
break: Entering Lisp.
1:> ans
(1 a b 2 (d) 3)
1:> (continue)
[ continue from break loop ]

yes
| ?- : '[a, b, ','bad, c].
error: unbound variable - bad
Lisp evaluation error trap!  Lisp received:
(backquote (a b (comma bad) c))
Prolog form:
[backquote,[a,b,[comma,bad],c]]

no
| ?- lisptrace(both).

yes
| ?- sumlist([1,2,3,5,7,11], Sum).
-- Lisp trace -- <receive>:
(apply (function +) (quote (1 2 3 5 7 11)))
-- Lisp trace -- <eval>:
```

```
29

Sum = 29

yes
| ?- lisptrace(none).

yes
| ?- halt.

[ Prolog execution halted ]
4 % exit
```

7. Bugs

7.1. IPL bugs

1. **:sin(1.0)** fails.

   The data conversion from Prolog to Lisp will convert integral floating-point numbers to integers. Thus **:sin(1.0)** will fail because 1.0 will be converted to 1, and the Lisp **sin** function requires a floating-point number. **(sin(1.1)** will work properly.) The Lisp **float** function can be used to work around this bug. For example, **:sin(float(1.0))** will work.

   This bug is not easily corrected because of C-Prolog's handling of integer and floating-point numbers:

   > In general, an arithmetic operation that combines integers and floating point numbers will return a floating point number. However, if the result is integral, it is converted back to integer representation, *and the same applies to numbers read in by the reader.* [Pereira,85,p23.]

2. IPL will probably ungracefully fail with very large integers or very large or small floating-point numbers. Integers within the range -268435456 to 268435455 (-(2^28) to (2^28)-1) are guaranteed to work.

3. Currently, the Lisp call predicates "fail" (in the Prolog goal sense) upon encountering a Lisp evaluation error. Some type of "abort" would probably be preferable. This wouldn't be difficult to implement — it requires a hack to xprolog.

   (Note: the C-Prolog predicate **abort** can't be used for this purpose because it closes all files, thus killing pxlisp!)

4. IPL improperly handles the reserved XLISP symbol ***unbound***. Workaround: Don't use the symbol, it's not portable anyway! Use the functions **boundp** and **makunbound**. (The latter is defined in init.lsp.)

7.2. C-Prolog bugs

1. Negative floating-point numbers are improperly handled. For example, try **write(-1.5)**. Remember this if you make a Lisp call that may return a negative floating point number.

7.3. XLISP 1.6 bugs

1. XLISP improperly handles "null" functions, e.g. **(defun x ())**. This in turn causes an IPL error. (This bug only appears in XLISP 1.6 on AT&T Unix PC's.)
2. XLISP writes all error messages to stdout, not stderr.

8. References

Betz, David Michael. *XLISP: An Experimental Object Oriented Language; Version 1.6.* January 6, 1986.

Pereira, Fernando (Editor); Warren, David; Bowen, David; Pereira, Luis. *C-Prolog User's Manual, Version 1.4.* April 24, 1985.

Recard, Steven J. *IPL: Interfaced Prolog/Lisp.* Master's Thesis, Rochester Institute of Technology, Rochester, NY, June 1987.

Missing Page

Missing Page

BIBLIOGRAPHY

Bailey, D. The University of Salford Lisp/Prolog System. *Software — Practice and Experience* 15,6 (June 1985), 595-609.

Balbin, Isaac; Lecot, Koenraad. *Logic Programming: a classified bibliography.* York Press, Abbotsford, Australia, 1985.

Barbuti, R.; Bellia, M.; Levi, G.; Martelli, M. On the Integration of Logic Programming and Functional Programming. *International Symposium on Logic Programming*, (Atlantic City, New Jersey, February 6-9). IEEE Computer Society Press, 1984.

Betz, David Michael. *XLISP: An Experimental Object Oriented Language; Version 1.6.* January 6, 1986.

Bowen, Kenneth A. New Directions in Logic Programming. In *1986 ACM Fourteenth Annual Computer Science Conference* (Cincinnati, Ohio, February 4-6), 1986, 19-22.

Boyer, R.S.; Moore, J.S. The Sharing of Structure in Theorem-proving Programs. In *Machine Intelligence 7*, Bernard Meltzer and Donald Michie, Editors. Wiley, New York, 1972.

Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

Brobow, Daniel G. If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms. *IEEE Transactions on Software Engineering* SE-11,11 (November 1985), 1401-1408.

Bruynooghe, Maurice. The Memory Management of Prolog Implementations. In *Logic Programming.* Clark, K.L.; Tarnlund, S.A. (Editors). Academic Press, New York, 1982.

Campbell, J.A. (Editor). *Implementations of Prolog.* Ellis Horwood, Pub. Wiley, New York, 1984.

Carlsson, Mats. On Implementing Prolog In Functional Programming. In *International Symposium on Logic Programming*, (Atlantic City, New Jersey, February 6-9). IEEE Computer Society Press, 1984.

Clark, K.L.; Tarnlund, S.A. (Editors). *Logic Programming.* Academic Press, New York, 1982.

Clocksin, W.F.; Mellish, C.S. *Programming in Prolog.* Second Edition, Springer-Verlag, New York, 1984.

Goto, Shigeki. DURAL: an extended Prolog language. In *Proceedings from the RIMS Symposia on Software Science and Engineering*, (Kyoto, 1982). *Lecture Notes in Computer Science* series, Edited by Goos, G.; Hartmanis, J. Springer-Verlag, New York, 1983.

Henderson, Peter. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

Hutchinson, S.A.; Kak, A.C. FProlog: A language to Integrate Logic and Functional Programming for Automated Assembly. In *Proceedings of the 1986 IEEE International Conference on Robotics and Automation* (San Francisco, California, April 7-10), 1986, 904-909.

Kahn, Kenneth M. UNIFORM—A Language based upon unification which unifies (much of) Lisp, Prolog, and Act 1. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (Vancouver, B.C., Canada, August 24-28). 1981, 933-939.

Kahn, Kenneth M.; Carlsson, M. How to implement Prolog on a LISP machine. In *Implementations of Prolog.* Campbell, J.A. (Editor). Ellis Horwood, Pub. Wiley, New York, 1984.

Kluzniak, Feliks; Szpakowicz, Stanislaw. *Prolog for Programmers.* Academic Press, New York, 1985.

Komorowski, H. Jan. QLOG—The Programming Environment for Prolog in Lisp. In *Logic Programming*, Clark, K.L.; Tarnlund, S.A. Eds. Academic Press, New York, 1982.

Kornfeld, William A. Equality for Prolog. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany, August 8-12). 1983, 514-519.

Kowalski, Robert. Algorithm = Logic + Control. *Communications of the ACM* 22,7 (July 1979), 424-436.

Kowalski, Robert. *Logic for Problem Solving.* Artificial Intelligence Series, The Computer Science Library. Elsevier Science Publishing, New York, 1983.

Lloyd, J.W. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

Malachi, Yonathan; Manna, Zohar; Waldinger, Richard. *TABLOG: A New Approach to Logic Programming.* Report No. STAN-CS-86-1110, Stanford University, Stanford, CA, Department of Computer Science, March 1985.

Mellish, Chris; Hardy, Steve. Integrating Prolog in the POPLOG environment. In *Implementations of Prolog.* Campbell, J.A. (Editor). Ellis Horwood, Pub. Wiley, New York, 1984.

Nilsson, M. The world's shortest Prolog interpreter? In *Implementations of Prolog.* Campbell, J.A. (Editor). Ellis Horwood, Pub. Wiley, New York, 1984.

O'Keefe, R.A. Prolog Compared with Lisp? *ACM SIGPLAN Notices* 18,5 (May 1983), 46-56.

Pereira, Fernando (Editor); Warren, David; Bowen, David; Pereira, Luis. *C-Prolog User's Manual, Version 1.4.* April 24, 1985.

Poe, M.D.; Nasr, R.; Potter, J.; Slinn, J. A KWIC (key word in context) bibliography on Prolog and logic programming. *Journal of Logic Programming* 1,1 (June 1984), 81-142.

Roach, John; Fowler, Glenn. Virginia Tech Prolog/Lisp—A Dual Interpreter Implementation. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, (Honolulu, Hawaii). 1985, pp. 88-92.

Robinson, J.A.; Sibert, E.E. LOGLISP: an alternative to PROLOG. In *Machine Intelligence 10*, J.E. Hayes, Editor. Ellis Horwood, Pub. Wiley, New York, 1982.

Sato, Masahiko; Sakurai, Takafumi. QUTE: A Prolog/Lisp Type Language for Logic Programming. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany, August 8-12). 1983, 507-513.

Steele, Guy L. Jr. *Common LISP — The Language.* Digital Press, Digital Equipment Corporation, USA, 1984.

Sterling, Leon. *The Art of Prolog.* MIT Press, 1986.

Subrahmanyam, P.A.; You, J-H. Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. In *International Symposium on Logic Programming*, (Atlantic City, New Jersey, February 6-9). IEEE Computer Society

Press, 1984.

Subrahmanyam, P.A. The "Software Engineering" of Expert Systems: Is Prolog Appropriate? *IEEE Transactions on Software Engineering* SE-11,11 (November 1985), 1391-1400.

Takeuchi, Ikuo; Okuno, Hiroshi; Ohsato, Nobuyasu. TAO—A harmonic mean of Lisp, Prolog, and Smalltalk. *ACM SIGPLAN Notices* 18,7 (July 1983), 65-74.

Warren, D.H.D.; Pereira, Luis; Pereira, Fernando. Prolog— The language and its implementation compared with Lisp. *ACM SIGPLAN Notices* 12,8 (August 1977), 109-115.

Warren, D.H.D. Higher-order extensions to Prolog: are they needed? *Machine Intelligence 10*, J.E. Hayes, Editor. Ellis Horwood, Pub. Wiley, New York, 1982.

Wilensky, Robert. *LISPcraft*. W.W. Norton & Company, New York, 1984.

Winston, Patrick Henry; Horn, Berthold. *LISP*, Second Edition. Addison-Wesley, Reading, Massachusetts, 1984.